21EC810 – PROJECT
Review - 2

Hybrid Reinforcement Learning for QOS Based Adaptive Routing in Hierarchical Software Defined Networks

PROJECT GUIDE : Dr.E.Murugavalli
PRESENTED BY: Sriram Vignesh V
(21D095)

Introduction:

Overview:

•Software-Defined Networking (SDN):

- Software-defined networks (SDNs) have been recognized as the next-generation networking paradigm that decouples the data forwarding from the centralized control..

Key Challenges:

•QoS Provisioning:

- Meeting diverse Quality-of-Service (QoS) requirements (e.g., delay, packet loss, throughput) is critical for modern applications.

•Scalability & Adaptation:

- As network size and traffic patterns change over time, fast route reconfiguration and adaptive routing are needed.
- Lack of global view and no broadcast allowed
- Statically allocated end to end links

Objectives of This Project:

Implement QoS-Aware Adaptive Routing Algorithm:
•Develop a hierarchical reinforcement learning-based routing strategy that dynamically adjusts to network congestion, delay while optimizing available bandwidth.
•Ensure efficient QoS provisioning by considering real-time link conditions in path selection.

Separation of Control and Data Planes:
•Design a multi-layer SDN control architecture comprising slave, domain, and super controllers for scalable, distributed decision-making.
•Enable domain controllers to manage local routing decisions while a super controller aggregates global network insights to refine policies.

Application of Reinforcement Learning (SARSA) for Routing:
•Implement the SARSA algorithm to enable adaptive, experience-based learning of optimal routes in SDN.
•Design a custom reward function incorporating delay, queuing delay, packet loss, and bandwidth utilization to guide efficient path selection.

Integration of Federated Learning in Hybrid RL-Based Routing with weight transfer:
•Allow domain controllers to compute local Q-table updates based on their network segment.
•Periodically transmit delta updates to the super controller, which aggregates these updates into a global Q-table without requiring direct access to all network data.
•Mimic federated learning principles to enhance scalability, reduce overhead, and adapt dynamically to network changes.

Specifications:

Software used : OMNET++ 6.0.2
Framework: INET 4.3.0
Programming languages required:

1) C++ (For INET framework)
2) .ned (network description language)
3) .ini (omnet++ network initialization language)
4) Python (For result analysis)
5) Shell scripts

Expected Outcomes:

Enhanced QoS-Aware Routing Performance:
•Improve network resilience by enabling adaptive routing that reacts to traffic congestions

•**QoS-Adaptive Routing with Adjustable Reward Function:**
•Deliver flexible routing decisions that dynamically balance QoS trade-offs (e.g., prioritizing delay-sensitive or bandwidth-intensive flows).
•Provide a tunable reward function, allowing network operators to adjust weights for different QoS parameters to suit specific network conditions and application requirements.

Hierarchical Control Plane Efficiency:
•Demonstrate the effectiveness of a multi-layer control plane, where slave controllers manage local switches, domain controllers handle regional updates, and a super controller oversees global policy decisions.
•Reduce overhead by ensuring local decision-making while leveraging global awareness for optimized routing.

Reinforcement Learning Convergence:
•Validate the SARSA-based RL model by analysing the convergence of the global Q-table showing that the RL agent learns an optimal routing policy over time, reducing instability and unnecessary route fluctuations.

Federated Learning-Based Routing Optimization:
•Enable domain controllers to compute local Q-table updates based on observed network conditions.
•Periodically share delta updates with the super controller, which aggregates them using federated learning-inspired mechanisms.
•Achieve a globally optimized routing policy without requiring full network visibility at each domain controller, improving scalability and privacy.

Action Plan:

Simulation Environment Setup:

1. Configure OMNeT++ simulation with the provided NED file and parameters (omnetpp.ini).
2. Integrate all components (sdn_switch, slave_controller, domain_controller, super_controller).

Hierarchical Controller Implementation:

1. Develop individual controllers:
   1. Slave Controllers: Collect and forward network condition data from switches.
   2. Domain Controllers: Perform local SARSA-based reinforcement learning for routing.
   3. Super Controller: Aggregate federated Q-table updates and compute the global route plan.

Reinforcement Learning Module Development:

1. Implement the SARSA algorithm in domain and super controllers.
2. Define a QoS-aware reward function based on delay, queuing delay, packet loss, and available bandwidth.
3. Use a softmax policy for action selection.

Federated Learning Integration:

1. Have domain controllers compute Q-table delta updates after each SARSA episode.
2. Transmit these updates (scaled by a federated learning rate) to the super controller.
3. Aggregate updates via simple averaging to refine the global Q table.

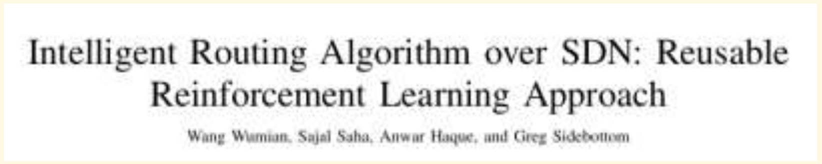
Parameter Tuning & Validation:

1. Fine-tune learning parameters (alpha, gamma, tau, federatedRate, etc.).
2. Compare against baseline routing strategies to validate improvements.

Documentation & Presentation:

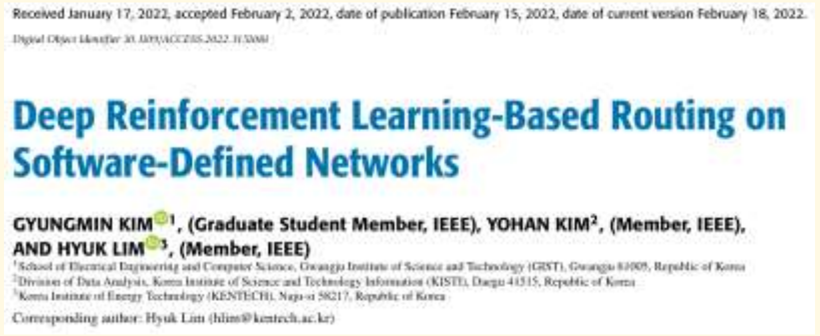
1. Prepare detailed documentation of the implementation.
2. Assemble a project review presentation (including simulation results and convergence graphs).

Reference papers:


Intelligent Routing Algorithm over SDN: Reusable Reinforcement Learning Approach

Wang Wumian, Sajal Saha, Anwar Haque, and Greg Sidebottom

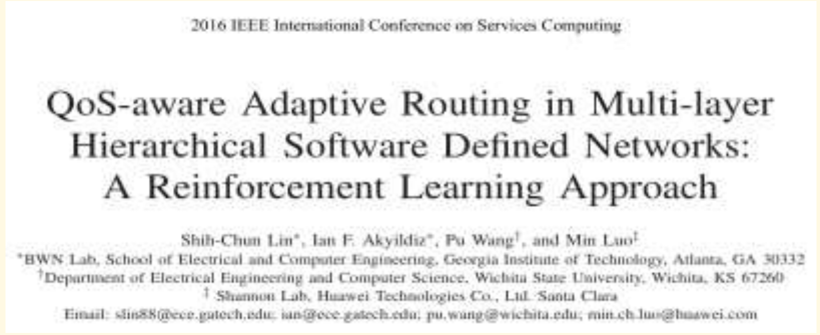
arXiv:2409.15226v1 [cs.NI] 23 Sep 2024
Year : 2024 publisher : Arxiv


Received January 17, 2022, accepted February 2, 2022, date of publication February 15, 2022, date of current version February 18, 2022.
Digital Object Identifier 10.1109/ACCESS.2022.3151081

Deep Reinforcement Learning-Based Routing on Software-Defined Networks

GYUNGMIN KIM[1], (Graduate Student Member, IEEE), YOHAN KIM[2], (Member, IEEE), AND HYUK LIM[3], (Member, IEEE)
[1]School of Electrical Engineering and Computer Science, Gwangju Institute of Science and Technology (GIST), Gwangju 61005, Republic of Korea
[2]Division of Data Analysis, Korea Institute of Science and Technology Information (KISTI), Daegu 41515, Republic of Korea
[3]Korea Institute of Energy Technology (KENTECH), Naju-si 58217, Republic of Korea
Corresponding author: Hyuk Lim (hlim@kentech.ac.kr)

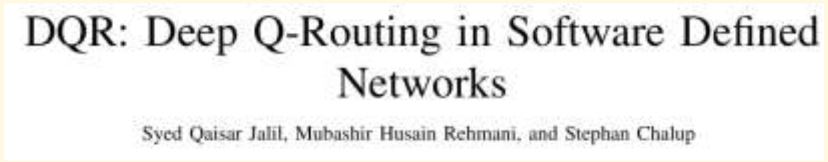
DOI 10.1109/ACCESS.2022.3151081
Year : 2022 Publisher : IEEE


2016 IEEE International Conference on Services Computing

QoS-aware Adaptive Routing in Multi-layer Hierarchical Software Defined Networks: A Reinforcement Learning Approach

Shih-Chun Lin*, Ian F. Akyildiz*, Pu Wang†, and Min Luo‡
*BWN Lab, School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332
†Department of Electrical Engineering and Computer Science, Wichita State University, Wichita, KS 67260
‡Shannon Lab, Huawei Technologies Co., Ltd. Santa Clara
Email: slin88@ece.gatech.edu; ian@ece.gatech.edu; pu.wang@wichita.edu; min.ch.luo@huawei.com

DOI 10.1109/SCC.2016.12
Year :2016 Publisher: IEEE


DQR: Deep Q-Routing in Software Defined Networks

Syed Qaisar Jalil, Mubashir Husain Rehmani, and Stephan Chalup

Doi: 10.1109/IJCNN48605.2020.9206767.
Year: 2019 Publisher: IEEE

Comparison table

Aspect	Paper 1: Intelligent Routing Algorithm (RLSR-Routing)	Paper 2: Deep RL-Based Routing	Paper 3: QoS-Aware Adaptive Routing (QAR)	Paper 4: Deep Q-Routing (DQR)	**Hybrid RL Routing with Federated Learning**
Approach	Reusable RL algorithm for QoS-aware routing	Deep Reinforcement Learning (DRL)	Adaptive RL with hierarchical SDN	Dueling Deep Q-Network	Hybrid SARSA-based RL with Federated weight transfer
QoS Parameters	Delay, Load Balancing	Delay, Packet Loss, Bandwidth	Delay, Packet Loss, Bandwidth	Delay, Bandwidth, Loss, Cost	Delay, Queueing Delay, Packet Loss, Bandwidth
Learning Method	Traditional RL (Reusable Q-learning)	Deep Q-Learning	Reinforcement Learning (Q-Learning)	Dueling Deep Q-Learning	Weight transfer between controllers and Hybrid RL
Network Architecture	Single-layer, SDN Controller	Single Controller	Multi-layer Hierarchical SDN	Single Controller	Multi-layer Hierarchical SDN (Slave, Domain, Super Controllers)
Novelty/Contribution	Loop-free path exploration, Reduced Controller-Switch communication overhead	Uses Deep Q-Learning for improved QoS routing	Time-efficient adaptive routing, Hierarchical controller architecture	Prioritized Experience Replay, Better Throughput	Faster and better learning , obtains better convergence and global optimal paths

Literature Review – 1

Title	Authors	Journal/Year	DOI/Link	Remarks
Intelligent Routing Algorithm over SDN: Reusable Reinforcement Learning Approach	Wang Wumian, Sajal Saha, Anwar Haque, Greg Sidebottom	arXiv/2024	arXiv:2409.15226v1	The paper introduces RLSR-Routing, a QoS-aware, reusable RL algorithm for SDN. It ensures loop-free path exploration and leverages segment routing to achieve flow-based, source packet routing, reducing controller-plane communication. The algorithm demonstrates improved load balancing and faster convergence compared to traditional methods.

Literature Review – 2

Title	Authors	Journal/Year	DOI/Link	Remarks
Deep Reinforcement Learning-Based Routing on Software-Defined Networks	Gyungmin Kim, Yohan Kim, and Hyuk Lim	IEEE Access, Published: February 15, 2022	10.1109/ACCESS.2022.3151081	This paper proposes a deep reinforcement learning (DRL)-based routing optimization method for SDNs. The DRL agent learns the interdependency between network switch traffic loads and overall network performance, aiming to balance end-to-end delay and packet loss

Literature Review – 3

Title	Authors	Journal/Year	DOI/Link	Remarks
QoS-Aware Adaptive Routing in Multi-layer Hierarchical Software Defined Networks: A Reinforcement Learning Approach	Shih-Chun Lin, Ian F. Akyildiz, Pu Wang, Min Luo	2016 IEEE International Conference on Services Computing	10.1109/SCC.2016.20	This paper proposes a QoS-aware adaptive routing (QAR) algorithm for multi-layer hierarchical SDNs. The architecture employs a distributed hierarchical control plane with super, domain, and slave controllers to minimize signalling delay in large networks. The QAR algorithm utilizes reinforcement learning with a QoS-aware reward function to achieve time-efficient, adaptive packet forwarding. Simulation results indicate that QAR outperforms existing learning solutions, providing fast convergence with QoS provisioning, facilitating practical implementations in large-scale software service-defined networks.

Literature review - 4

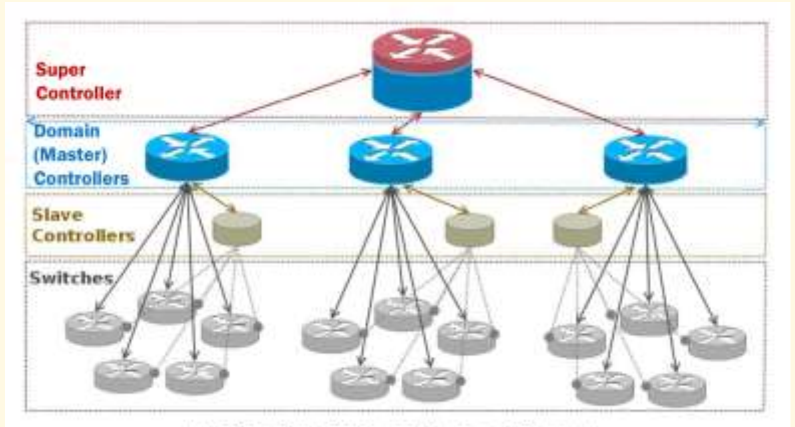
Title	Authors	Conference/Year	DOI/Link	Remarks
DQR: Deep Q-Routing in Software Defined Networks	Syed Qaisar Jalil, Mubashir Husain Rehmani, Stephan Chalup	2020 International Joint Conference on Neural Networks (IJCNN), Published: July 2020	10.1109/IJCNN48605.2020.9206767	This paper introduces Deep Q-Routing (DQR), a deep reinforcement learning solution for QoS routing in SDNs. DQR employs a dueling deep Q-network with prioritized experience replay to determine paths for source-destination pairs, considering multiple QoS metrics such as delay, bandwidth, loss, and cost. The approach treats routing as a discrete control problem and utilizes a reward function comprising weighted QoS parameters. Simulation results indicate that DQR substantially improves end-to-end throughput compared to existing learning-based methods.

Note: The paper was presented at the 2020 International Joint Conference on Neural Networks (IJCNN).

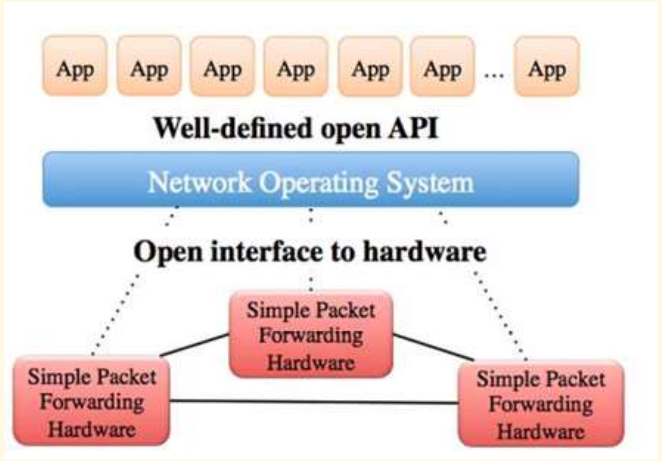
Design Methodology

Hierarchical Control Architecture:

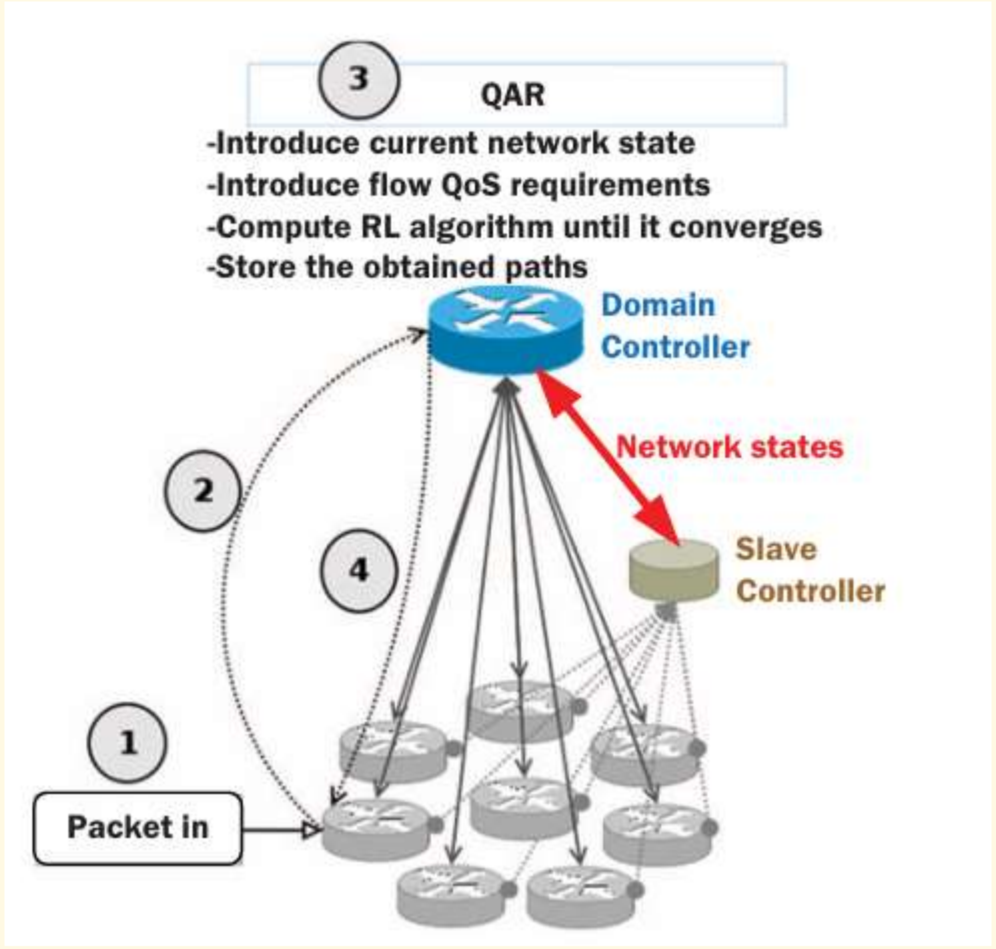
- **Slave Controllers:**
  - Collect real-time network metrics (delay, queuing, loss, bandwidth) from switches.
  - Forward raw condition data to the domain controllers.
- **Domain Controllers:**
  - Process network condition data locally.
  - Execute a SARSA-based reinforcement learning algorithm using a custom QoS-aware reward function.
  - Generate local Q-tables and compute delta updates (difference from a stored Q_prev).
- **Super Controller:**
  - Receives federated (delta) updates from domain controllers.
  - Aggregates updates (using federated averaging) to maintain a global Q-table.
  - Disseminates the updated routing plan back to the network.

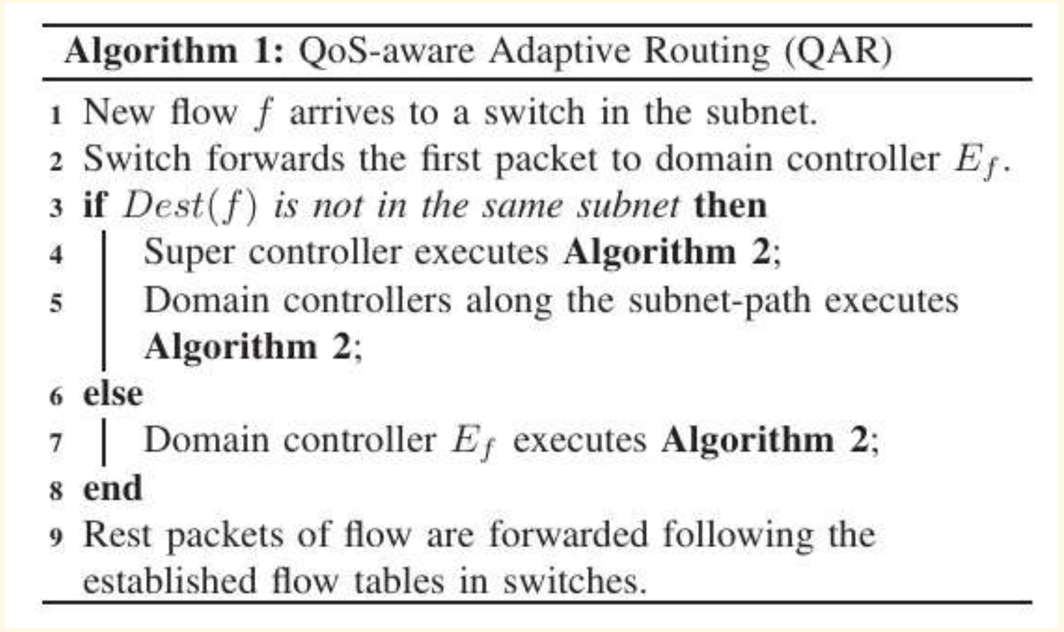


(a) Distributed hierarchical architecture.

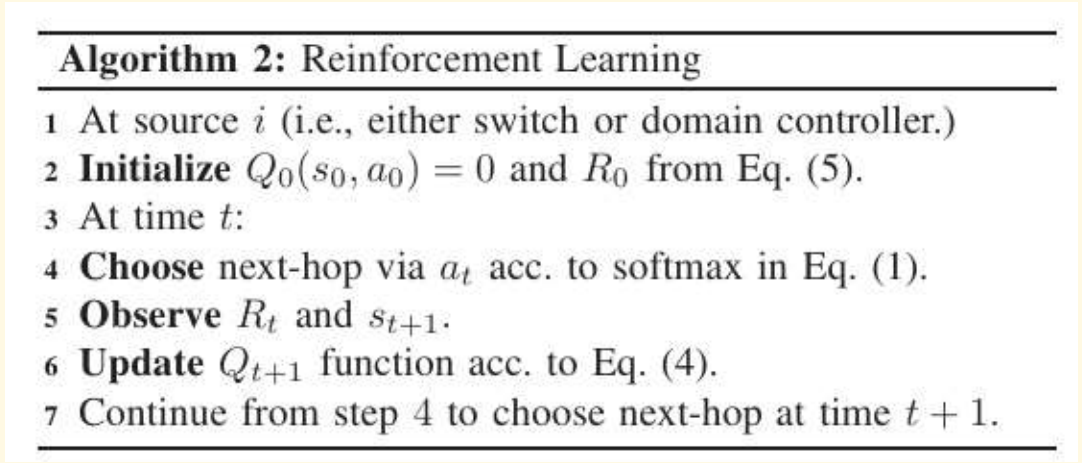

App
App
App
App
App
App
...
App
Well-defined open API
Network Operating System
Open interface to hardware
Simple Packet Forwarding Hardware
Simple Packet Forwarding Hardware
Simple Packet Forwarding Hardware

QAR routing with RL:


3
QAR
-Introduce current network state
-Introduce flow QoS requirements
-Compute RL algorithm until it converges
-Store the obtained paths
Domain Controller
Network states
2
4
Slave Controller
1
Packet in

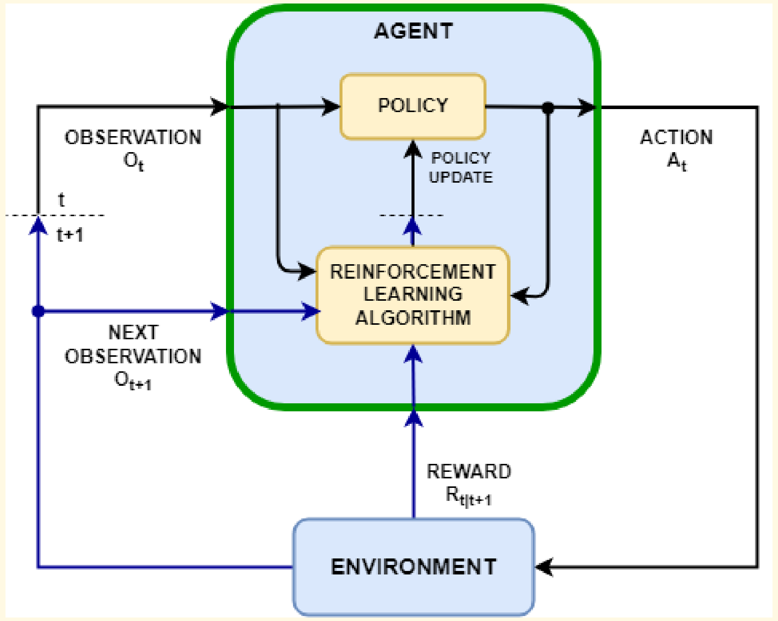


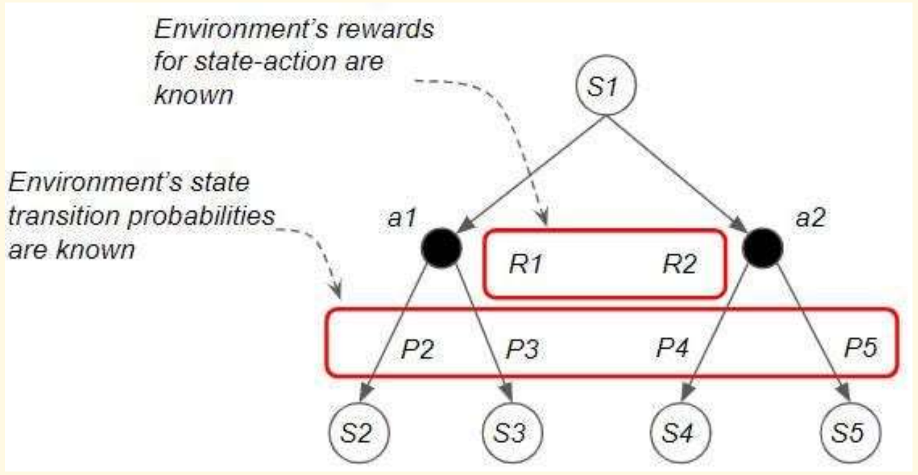
```
Algorithm 1: QoS-aware Adaptive Routing (QAR)
1 New flow $f$ arrives to a switch in the subnet.
2 Switch forwards the first packet to domain controller $E_f$.
3 if $Dest(f)$ is not in the same subnet then
4     Super controller executes Algorithm 2;
5     Domain controllers along the subnet-path executes
      Algorithm 2;
6 else
7     Domain controller $E_f$ executes Algorithm 2;
8 end
9 Rest packets of flow are forwarded following the
  established flow tables in switches.
```



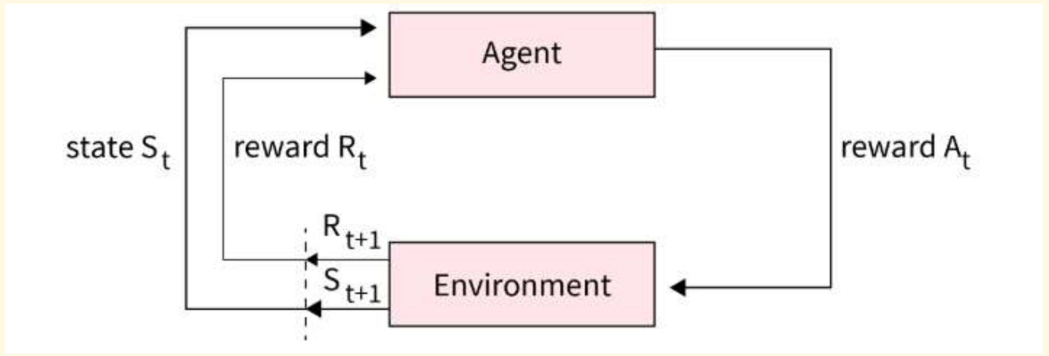
```
Algorithm 2: Reinforcement Learning
1 At source $i$ (i.e., either switch or domain controller.)
2 Initialize $Q_0(s_0, a_0) = 0$ and $R_0$ from Eq. (5).
3 At time $t$:
4 Choose next-hop via $a_t$ acc. to softmax in Eq. (1).
5 Observe $R_t$ and $s_{t+1}$.
6 Update $Q_{t+1}$ function acc. to Eq. (4).
7 Continue from step 4 to choose next-hop at time $t + 1$.
```

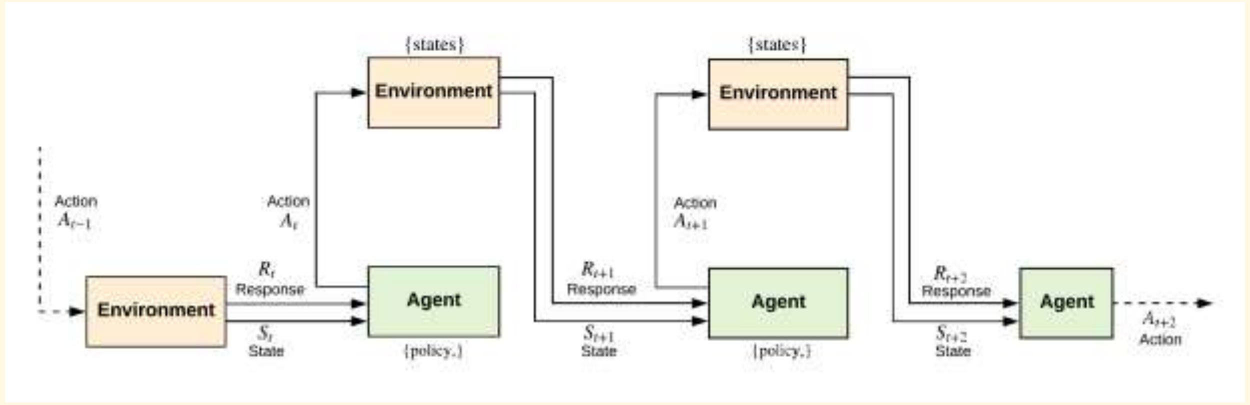
REINFORCEMENT LEARNING:




Environment's rewards for state-action are known
Environment's state transition probabilities are known
S1
a1
a2
R1
R2
P2
P3
P4
P5
S2
S3
S4
S5

MARKOV DECISION PROCESS USED IN RL (SARSA) :

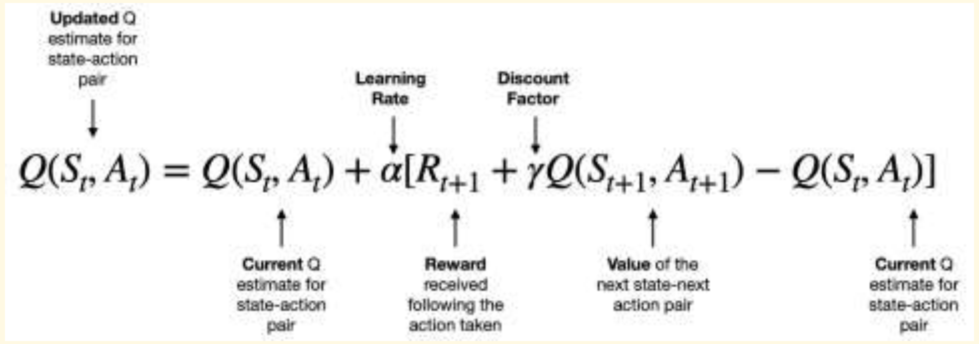




A Markov Decision Process can be represented by the tuple (S,A,P,R,γ)
Where,

- S refers to the set of states in the environment.
- A denotes the set of possible actions that the agent can choose.
- *P* represents the probability distribution of the next state when the current state and chosen action are given.
- *R* refers to the reward the agent obtains after choosing an action at its current state.
- γ refers to the discount factor used to compute discounted future rewards.

THE SARSA ALGOTITHM FOR RL:


$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$
Updated Q estimate for state-action pair
Learning Rate
Discount Factor
Current Q estimate for state-action pair
Reward received following the action taken
Value of the next state-next action pair
Current Q estimate for state-action pair

- $Q_{t+1}(s_t, a_t) := Q_t(s_t, a_t) + \alpha [R_t + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)]$

- $Q_t(s_t, a_t)$ – The current Q-value for state s_t and action a_t, representing the estimated reward when taking action a_t in state s_t.
- α – The learning rate, which determines how much new experiences influence the Q-value update.
- R_t – The immediate reward obtained at time step t after taking action a_t, based on QoS
- γ – The discount factor, which balances the importance of immediate versus future rewards.
- $Q_t(s_{t+1}, a_{t+1})$ – The Q-value for the next state s_{t+1} and the next action a_{t+1}; it estimates the future rewards from that state–action pair.
- $Q_t(s_t, a_t)$ – Subtracted again to adjust the Q-value based on the difference between the predicted and current estimates.

Q learning policy (off policy / epsilon -greedy) vs SARSA:

1. Q-Learning (Off-Policy)

•**Equation:**

$Q(s,a) \leftarrow Q(s,a) + \alpha\, [r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$

•**Key Features:**

- Learns optimal policy **independent** of the agent’s actual actions (Off-Policy).
- Uses the **greedy action** ($\max_{a'}$ $Q(s',a')$) for future state updates.
- Encourages faster convergence but may take riskier paths.

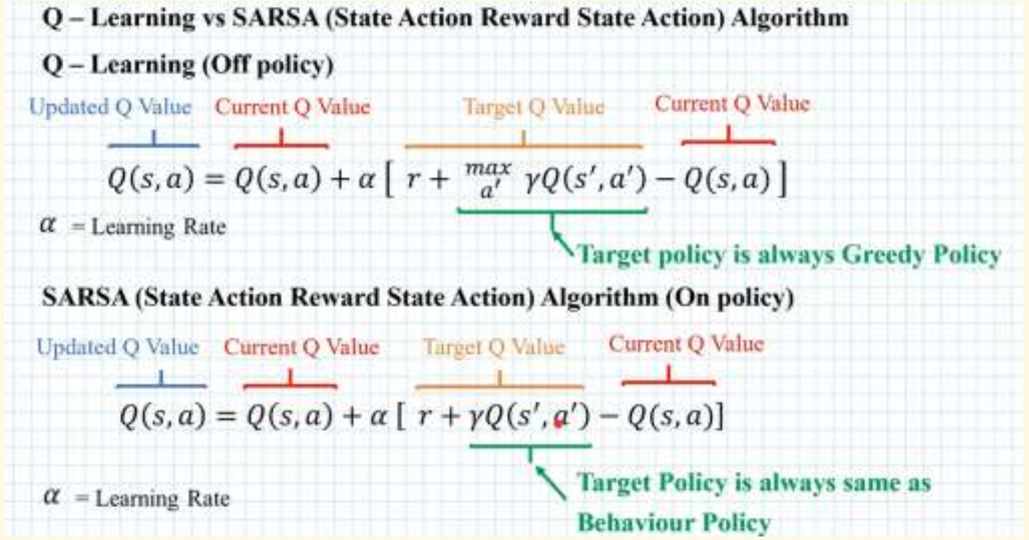
2. SARSA (On-Policy)

•**Equation:**

$Q(s,a) \leftarrow Q(s,a) + \alpha\, [r + \gamma\, Q(s',a') - Q(s,a)]$

•**Key Features:**

- Learns the policy **while following it** (On-Policy).
- Uses the **next selected action a'** (instead of $\max_{a'}$ $Q(s',a')$).
- Safer and more stable in QoS-sensitive applications, ensuring controlled learning.
- SARSA is preferred due to its smooth learning curve, making it **more stable for real-time QoS-aware routing in SDN.**
- Helps in **adaptive route selection** by considering bandwidth, delay, and packet loss **while avoiding risky paths.**



Q – Learning vs SARSA (State Action Reward State Action) Algorithm

Q – Learning (Off policy)

$$\overbrace{Q(s,a)}^{\text{Updated Q Value}} = \overbrace{Q(s,a)}^{\text{Current Q Value}} + \alpha\, [\, \overbrace{r + \underbrace{\max_{a'} \gamma Q(s',a')}_{\text{Target policy is always Greedy Policy}}}^{\text{Target Q Value}} - \overbrace{Q(s,a)}^{\text{Current Q Value}} \,]$$

α = Learning Rate

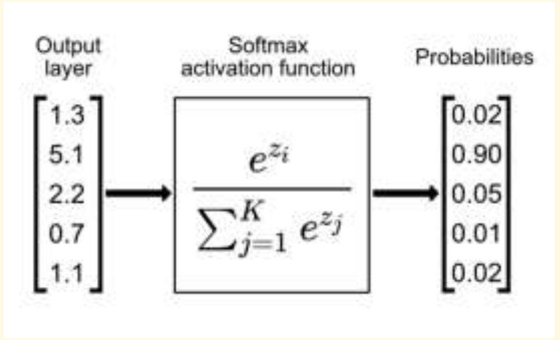
SARSA (State Action Reward State Action) Algorithm (On policy)

$$\overbrace{Q(s,a)}^{\text{Updated Q Value}} = \overbrace{Q(s,a)}^{\text{Current Q Value}} + \alpha\, [\, \overbrace{r + \underbrace{\gamma Q(s',a')}_{\text{Target Policy is always same as Behaviour Policy}}}^{\text{Target Q Value}} - \overbrace{Q(s,a)}^{\text{Current Q Value}}]$$

α = Learning Rate

POLICY FUNCTION (SOFTMAX) :

Formula: $P(a_i) = e^{(Q(s,a_i)/\tau)} / \sum_j e^{(Q(s,a_j)/\tau)}$


Output layer
Softmax activation function
Probabilities
$\begin{bmatrix}1.3\\5.1\\2.2\\0.7\\1.1\end{bmatrix} \rightarrow \boxed{\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}} \rightarrow \begin{bmatrix}0.02\\0.90\\0.05\\0.01\\0.02\end{bmatrix}$

- $P(a_i)$ -- Probability of selecting action a_i (choosing a routing path).
- $Q(s, a_i)$ -- Q-value representing expected reward based on QoS metrics (bandwidth, delay, packet loss).
- τ (Temperature) -- Balances exploration and exploitation (higher $\tau \rightarrow$ more exploration, lower $\tau \rightarrow$ more greedy selection).
- $e^{(Q(s,a_i)/\tau)}$ -- Assigns higher probabilities to actions with better Q-values while ensuring all actions have a chance.
- $\sum_j e^{(Q(s,a_j)/\tau)}$ -- Normalizes probabilities across all possible actions.

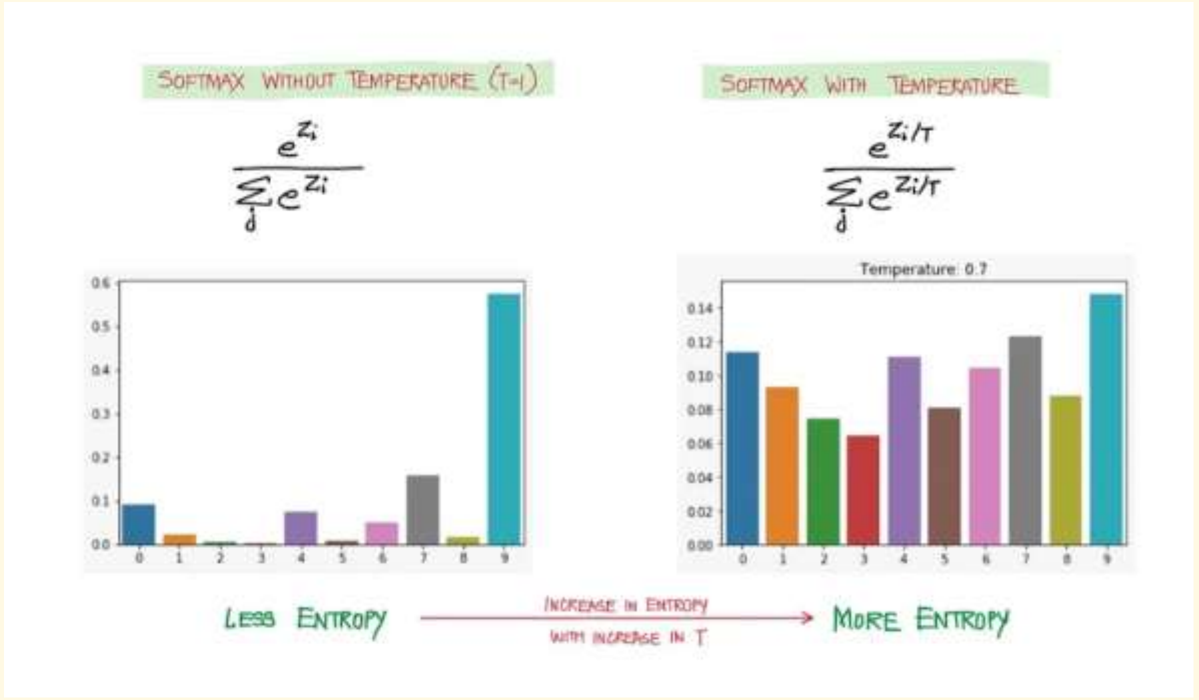
TEMPERATURE FUNCTION

Formula:

$\tau⍰ = -(\tau_o - \tau⍰)(n / T) + \tau_o$, for $n \leq T$

Explanation of Terms:

- **$\tau⍰$** – Temperature at iteration **n**, controlling exploration vs. exploitation.
- **τ_o** – Initial temperature, setting the starting randomness in action selection.
- **$\tau⍰$** – Final temperature, defining the level of exploitation at the end of training.
- **n** – Current iteration, representing the step in reinforcement learning.
- **T** – Total iterations over which temperature is gradually reduced.



Reward Function:

Reward Function:

$$R_t := R(i \to j \mid s_t, a_t) = -g(a_t) + \beta_1(\theta_1 \cdot delay_{ij} + \theta_2 \cdot queue_j) + \beta_2 \cdot loss_j + \beta_3(\phi_1 \cdot B1_{ij} + \phi_2 \cdot B2_{ij})$$

Explanation of Terms:

- R_t → Reward at time step t for selecting action a_t in state s_t.
- $g(a_t)$ → Penalty for action a_t (discourages frequent path changes).
- $\beta_1, \beta_2, \beta_3$ → Weighting factors for different QoS parameters (tunable for optimization).
- θ_1, θ_2 → Scaling factors that adjust the impact of *delay* and *queue* delay on the reward function.
- $delay_{ij}$ → Transmission delay between nodes i and j (higher delay leads to lower reward).
- $queue_j$ → Queueing delay at node j, indicating network congestion (higher queueing delay reduces reward).
- $loss_j$ → Packet loss rate at node j, which negatively impacts QoS (lower is better).
- ϕ_1, ϕ_2 → Scaling factors for available bandwidth parameters ($B1_{ij}$, $B2_{ij}$), controlling the impact of bandwidth on reward.
- $B1_{ij}$, $B2_{ij}$ → Available bandwidth measures between nodes i and j. These terms ensure load balancing by promoting paths with sufficient bandwidth.

QOS reward parameters:

i) $delay_{ij} = (2/\pi)\ \arctan(d_{ij}^{l} - (\sum_{k=1}^{A(i)} d_{ik}^{l}) / A(i));$

ii) $queue_{ij} = (2/\pi)\ \arctan(d_{ij}^{\psi} - (\sum_{k=1}^{N} d_{ik}^{\psi}) / N);$

iii) $loss_{ij} = 1 - 2\%\ loss_{ij};$

iv) $B1_{ij} = (2BW_{ij}^{A}) / BW_{ij}^{T} - 1;$

v) $B2_{ij} = (2/\pi)\ \arctan(0.01\ (BW_{ij}^{A} - (\sum_{k=1}^{N} BW_{ik}^{A}) / N));$

QOS reward parameters:

$$delay_{ij} = (2/\pi)\ \arctan(d_{ij}^{l} - (\sum_{\Box=1}^{A(i)} d_{i\Box}^{l}) / A(i))$$

Explanation of the Delay Calculation in QoS-Aware Routing:
The delay metric here is computed as:
$delay_{ij} = (2/\pi)\ \arctan(d_{ij}^{l} - (\sum_{\Box=1}^{A(i)} d_{i\Box}^{l}) / A(i))$

Where:
- d_{ij}^{l} → Direct link delay between nodes i and j.
- $A(i)$ → Number of alternative paths available at node i.
- $\sum_{\Box=1}^{A(i)} d_{i\Box}^{l} / A(i)$ → Average delay of alternative paths from i.

1. Delay Normalization with arctan:
    1. The arctan function is used to scale and normalize the delay values.
    2. This ensures that extremely large delays do not disproportionately impact routing decisions.
2. Balancing Direct Link Delay vs. Alternative Paths:
    1. The equation adjusts the direct link delay d_{ij}^{l} by subtracting the average delay of other available paths.
    2. If the selected path has higher than average delay, the function penalizes it to encourage the selection of lower-latency routes.
3. Adaptive Routing Behaviour:
    1. If a link consistently experiences high delay, the algorithm prefers alternate paths that have a lower average delay.
    2. This helps in load balancing by preventing overuse of congested links.

QOS reward parameters:

$loss_{ij} = 1 - 2 * \% \ loss_{ij}$

•This packet loss function adjusts the reward penalty based on observed loss rates.
•By subtracting twice the percentage loss, the function ensures that paths with higher reliability (lower loss) are preferred.
•In QoS-aware routing, packet loss is a critical metric affecting reliability.
•The 20% scaling factor ensures that even small variations in packet loss significantly influence routing decisions, but not excessively.

$queue_{ij} = (2/\pi) \arctan(d_{ij}^{\psi} - (\sum_{\square=1}^{N} d_{i\square}^{\psi}) / N)$

•Queueing delay represents the time packets spend waiting in buffers before transmission.
•The normalization using the arctan function ensures that extreme queueing delays are appropriately penalized.
•The delay at a specific node j is compared against the average queueing delay in the network (N) to balance load across multiple paths.

QOS reward parameters:

$B1_{ij} = (2BW_{ij}^{A}) / BW_{ij}^{T} - 1$

- Available bandwidth (BW_{ij}^{A}) is normalized with respect to the total bandwidth (BW_{ij}^{T}) to dynamically balance traffic loads.
- The result ensures that paths with higher bandwidth availability receive higher rewards in the SARSA-based routing algorithm.

Explanation of Terms:

- BW_{ij}^{A} (Available Bandwidth) → The currently available bandwidth on the link between nodes i and j.
- BW_{ij}^{T} (Total Bandwidth Capacity) → The maximum possible bandwidth on the same link.

1. Normalization for Consistent Scaling:
    1. The equation normalizes the available bandwidth relative to the total bandwidth.
    2. This ensures that different network links are compared on the same scale, independent of absolute capacity.
2. Value Ranges Between -1 and 1:
    1. When $BW_{ij}^{A} = BW_{ij}^{A}$ (full bandwidth available), $B1_{ij} = 1$ (maximum positive impact on reward).
    2. When $BW_{ij}^{A} = 0$ (fully congested link), $B1_{ij} = -1$ (negative impact, discouraging selection).
    3. This ensures balanced reinforcement learning rewards, preventing over-reliance on a single route.

QOS reward parameters:

$B2_{ij} = (2/\pi)\ \arctan(0.1\ (BW_{ij}^{A} - (\sum_{\square=1}^{N} BW_{i\square}^{A})\ /\ N))$

•This function accounts for bandwidth fairness across the network.
•If a specific link's bandwidth is much lower than the average network bandwidth, it gets penalized to discourage congestion-prone paths.
•The arctan function smoothens this penalty to prevent extreme variations in decision-making.

Explanation of Terms:
•BW_{ij}^{A} (Available Bandwidth on Link $i \rightarrow j$) → Measures how much bandwidth is free on a specific link.
•$\sum_{\square=1}^{N} BW_{i\square}^{A}\ /\ N$ (Average Bandwidth Across All Links) → Represents the mean available bandwidth across the network.
•arctan Function → Used to smoothen the impact of bandwidth variations.

1. Encourages Bandwidth Fairness:
    1. If a link's available bandwidth BW_{ij}^{A} is lower than the network's average, its reward is reduced to discourage excessive routing through that link.
    2. Conversely, if BW_{ij}^{A} is higher than the average, the reward increases, making the link preferable for routing.
2. Smooth Transition Using arctan:
    1. The arctan function ensures gradual reward changes, avoiding sudden penalty spikes that could destabilize routing decisions.
    2. This prevents excessive route switching, maintaining network stability.
3. Traffic Load Balancing:
    1. The 0.1 scaling factor ensures that minor fluctuations in bandwidth do not lead to drastic changes.
    2. This prevents congestion-prone links from being overused while still allowing adaptive decision-making.
4. Prevents Unfair Resource Allocation:
    1. If a particular link gets overloaded, this function ensures that it does not continue to be selected unfairly.
    2. The routing algorithm shifts traffic toward less congested paths, ensuring QoS-aware adaptive routing.

Hybrid RL Update Mechanism (global) :

- Delta Computation:
  - Domain controllers calculate the difference between the current Q-table and a stored Q_prev.
  - The delta is scaled by a federated learning rate.
- Global Aggregation:
  - The super controller receives these delta updates and averages them with its current global Q-table.
  - This approach maintains a global view without each domain controller needing full network visibility.

$$\Delta Q(i, j, k) = \text{federatedRate} \times (Q\_local(i, j, k) - Q\_prev(i, j, k))$$

Then, at the super controller, the global Q-value is updated by averaging its current value with the received delta:

$$Q\_global(i, j, k) \leftarrow (Q\_global(i, j, k) + \Delta Q(i, j, k)) * \text{averaging weight factor}$$

Explanation:

- Q_local(i, j, k): The Q-value learned by the domain controller in its local update (after running SARSA).
- Q_prev(i, j, k): The previously stored Q-value at the domain level, used as a reference.
- federatedRate: A scaling factor in the range [0,1] that controls how much of the local delta should be integrated into the global model.
- Averaging Step: The super controller then updates the global Q-table by taking the average of its current Q-value and the scaled delta.

Omnet++ 6.02

OMNeT++ is a modular, component-based C++ simulation library and framework, primarily for building network simulators it is public-source, modular and open-architecture simulation environment with strong GUI support and an embeddable simulation kernel. Its primary application area is the simulation of communication networks.

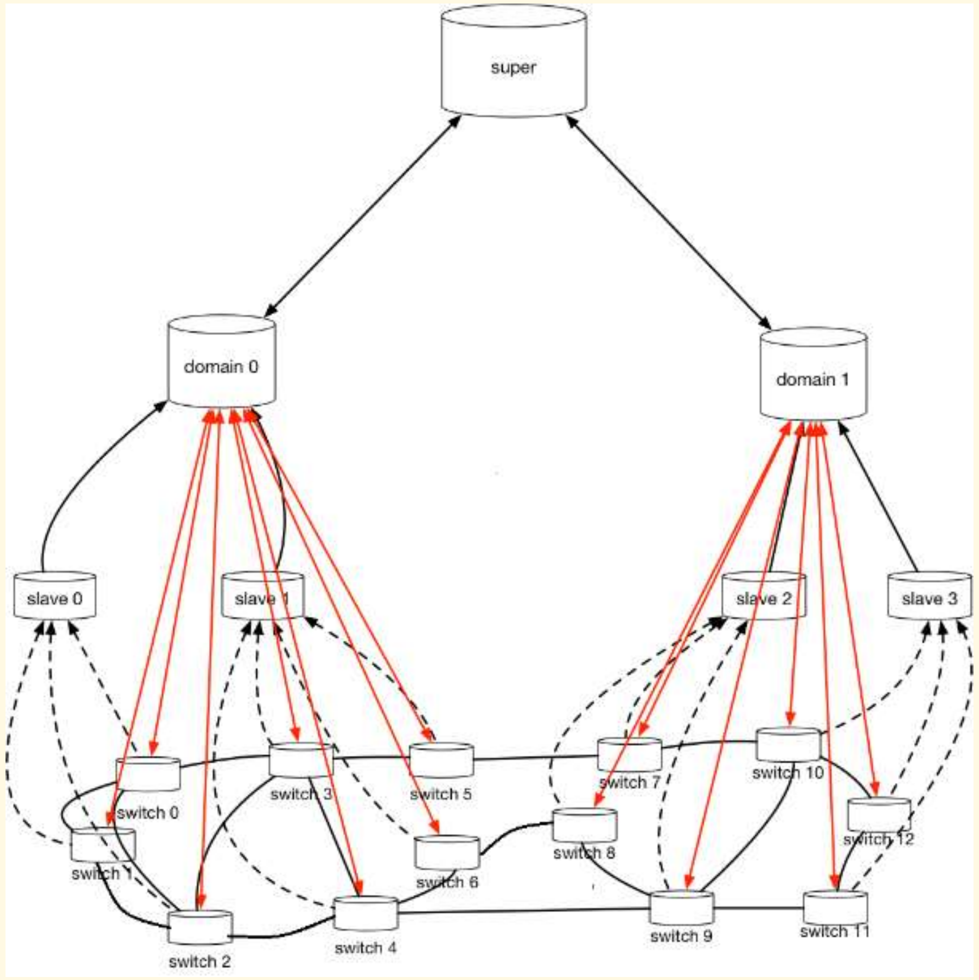
PROJECT STRUCT FILES IN OMNET++:

------ initialization files
------ result analysis files
------- core files for functionality

```
project_root/

├── omnetpp.ini               # Simulation parameters (random seeds, module configs, etc.)
├── slave_controller.cc       # Receives switch data, forwards condition info to domain
├── switch_message_m          # Auto-generated message class definitions for Switch <-> Controller
├── plot.py                   # Python script for plotting Q-table or convergence data
├── condition.h               # cObject storing link conditions (loss, delay, bandwidth, etc.)
├── switch.cc                 # SDN switch module (handles packets, forwards to domain or other switches)
├── domain_controller.cc      # Domain-level RL (SARSA) logic; updates local Q-table
├── route.h                   # Defines a Route object (stores next-hop info)
├── node.h                    # Helper struct for RL (e.g., storing node indices, values)
├── Q_table.txt               # Global Q-table data file (used by super controller)
├── domain_1_Q_table.txt      # Domain 1’s Q-table file
├── domain_2_Q_table.txt      # Domain 2’s Q-table file
├── super_controller.cc       # Super-level RL aggregator (federated updates, global Q-table)
├── SDN.ned                   # network description
```

SDN HIEARCHIAL STRUCTURE IMPLEMENTED:



-> Super controllers

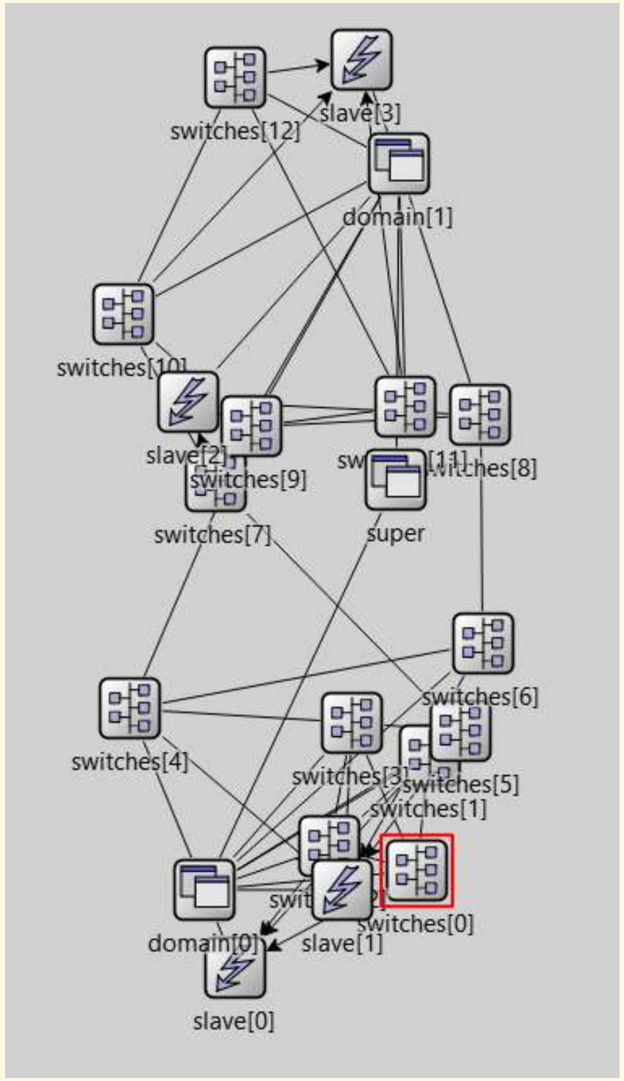
-> Domain controllers for subnet control

-> Slave controllers

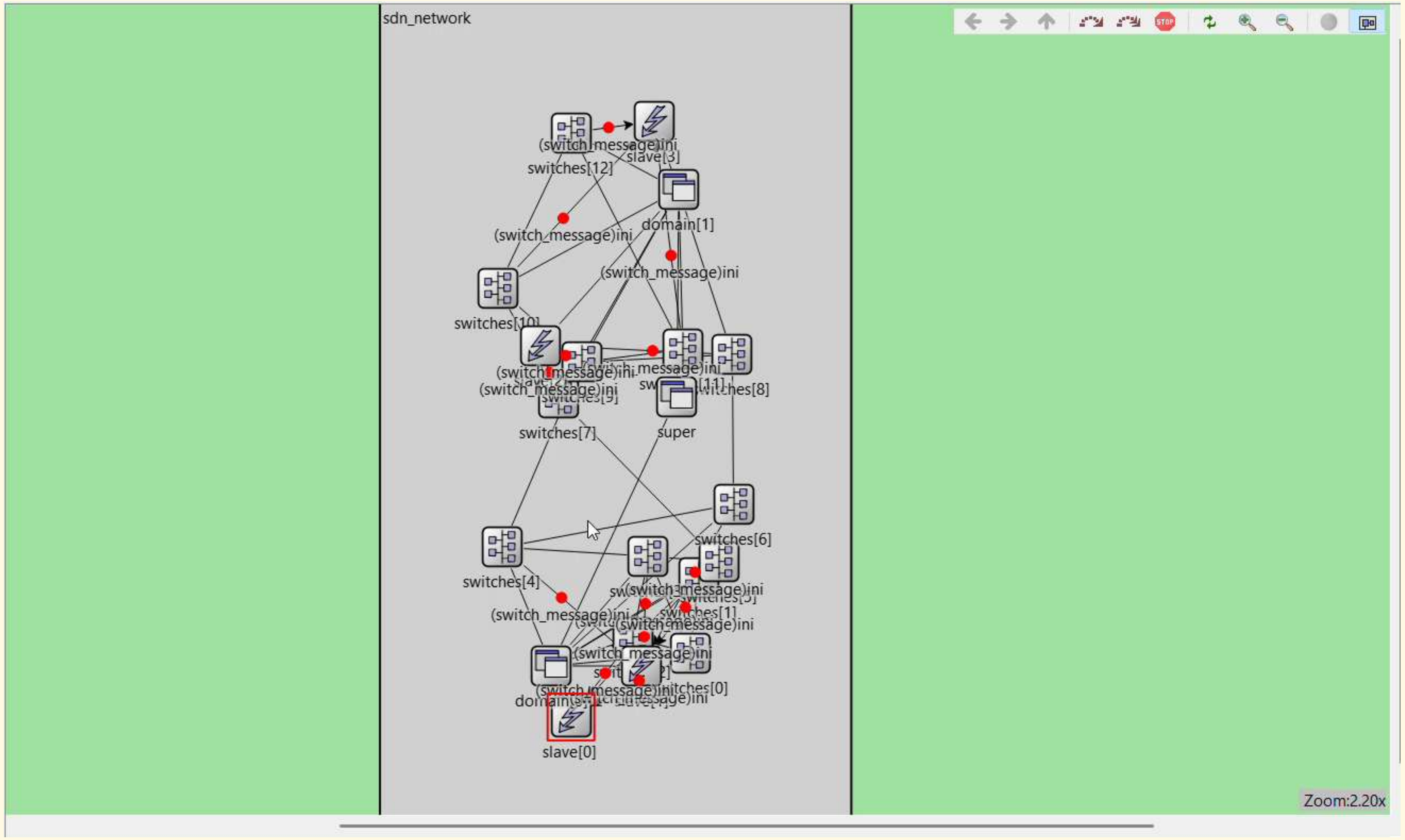
-> single duplex links for condition retrieval

-> end devices

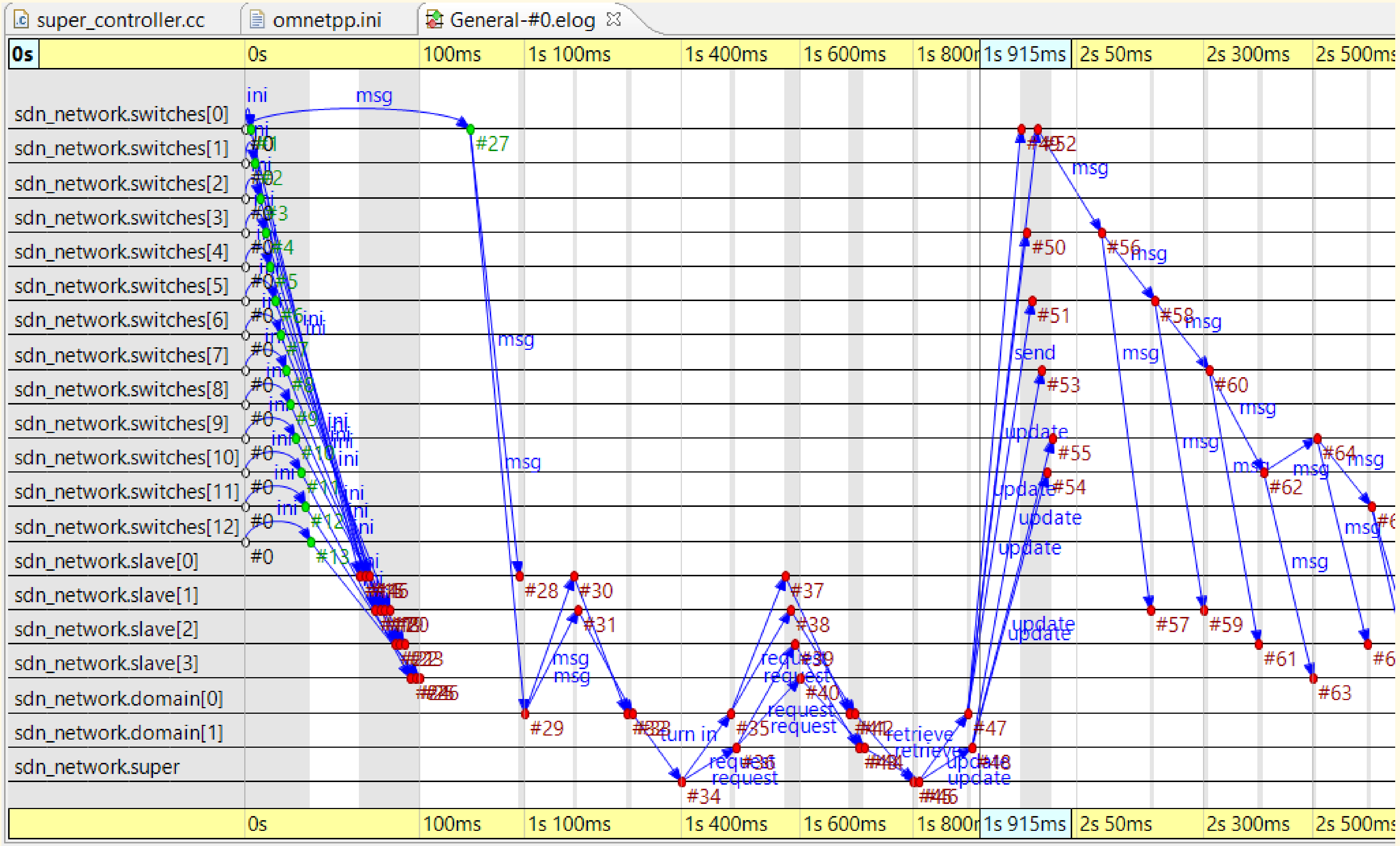
IMPLEMENTED TOPOLOGY IN OMNET++ 6.02:



WORKING DEMO FOR SINGLE QAR RUN:



Network state diagram for a single QAR run:



Sample Q-Values Learned for 250 Episodes On node 12 and 11

```
Q[i][j][k]
j -> source
i -> destination
k -> next hop Q-value
```

Is used in:

$Q_{t+1}(s_t, a_t) := Q_t(s_t, a_t) + \alpha [R_t + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)]$

Is converted to probabilities through policy function:

$P(a_i) = e\hat{}(Q(s,a_i)/\tau) / \sum_j e\hat{}(Q(s,a_j)/\tau)$

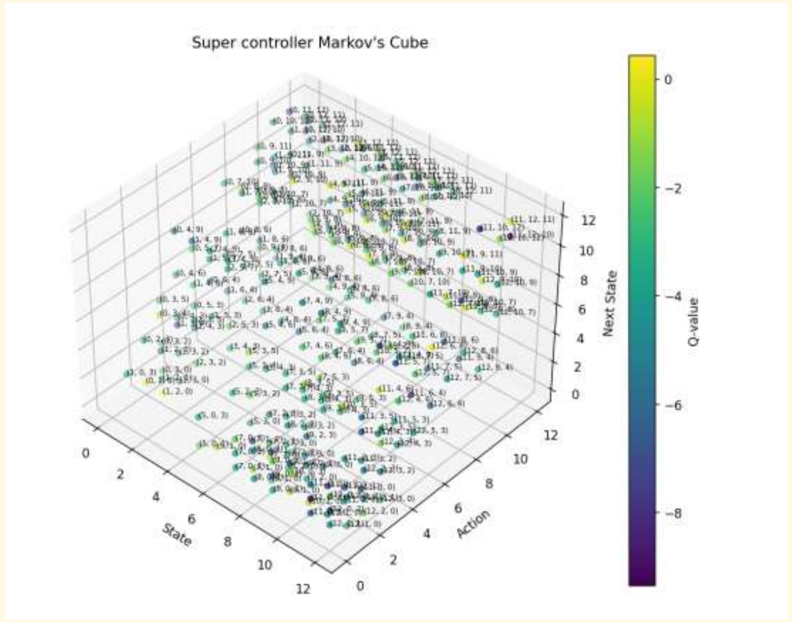
```
Q[12][0][1] = -4.17129
Q[12][0][2] = -6.08472
Q[12][0][3] = -2.00543
Q[12][1][0] = -2.54266
Q[12][1][2] = -5.31743
Q[12][2][0] = -1.8187
Q[12][2][1] = -3.89902
Q[12][2][3] = -5.14339
Q[12][3][0] = -2.43742
Q[12][3][2] = -5.04195
Q[12][3][4] = -1.43781
Q[12][3][5] = -2.71255
Q[12][4][3] = -4.33758
Q[12][4][6] = -2.48587
Q[12][4][9] = -4.7198
Q[12][5][3] = -4.1615
Q[12][5][7] = -2.20007
Q[12][6][4] = -5.81559
Q[12][6][8] = 0.103241
Q[12][7][5] = -4.65342
Q[12][7][10] = -1.78943
Q[12][8][6] = -2.87206
Q[12][8][9] = -0.245198
Q[12][9][4] = -3.48822
Q[12][9][8] = -2.60366
Q[12][9][10] = -0.403288
Q[12][10][7] = -2.65978
Q[12][10][9] = -4.22718
Q[12][10][12] = -1.50473
```

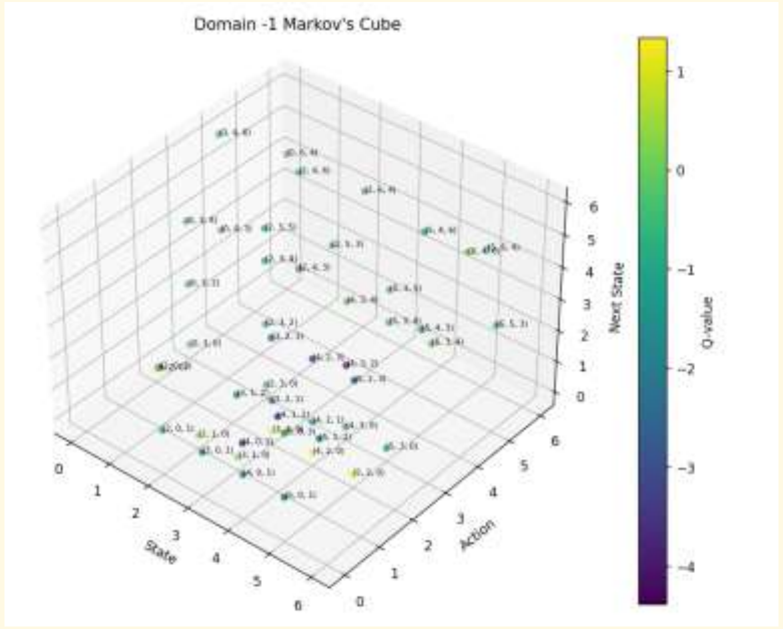
```
Q[11][0][1] = -6.23216
Q[11][0][2] = -8.53281
Q[11][0][3] = -5.34181
Q[11][1][0] = -5.93798
Q[11][1][2] = -7.56635
Q[11][2][0] = -2.98489
Q[11][2][1] = -6.94323
Q[11][2][3] = -4.87994
Q[11][3][0] = -7.01902
Q[11][3][2] = -4.62344
Q[11][3][4] = -6.90616
Q[11][3][5] = -5.6426
Q[11][4][3] = -5.72536
Q[11][4][6] = -0.562492
Q[11][4][9] = -7.08645
Q[11][5][3] = -4.49548
Q[11][5][7] = -6.75548
Q[11][6][4] = -8.18021
Q[11][6][8] = -2.40043
Q[11][7][5] = -4.75914
Q[11][7][10] = -3.91975
Q[11][8][6] = -6.25021
Q[11][8][9] = -0.596547
Q[11][9][4] = -6.08151
Q[11][9][8] = -7.12682
Q[11][9][10] = -3.81277
Q[11][9][11] = -0.0369277
Q[11][10][7] = -1.91295
Q[11][10][9] = -0.74383
Q[11][10][12] = -8.60144
Q[11][12][10] = -9.34128
Q[11][12][11] = -0.526529
```

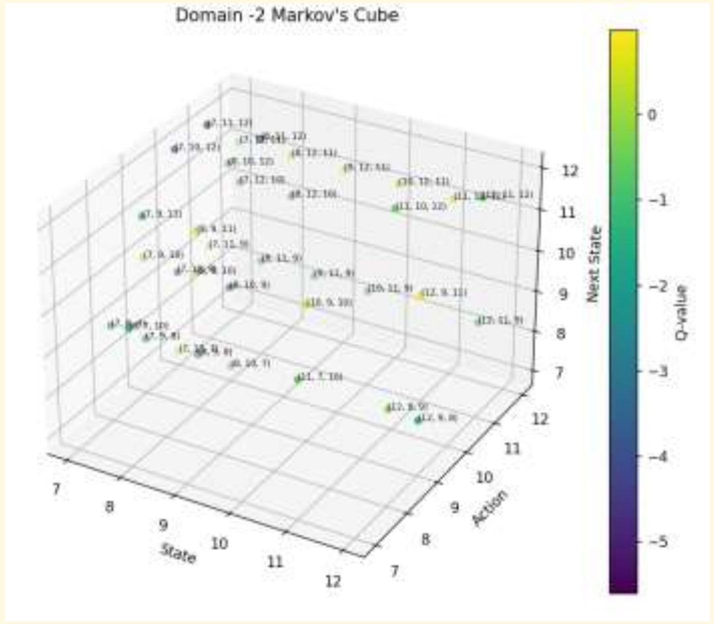
SAMPLE LOG FILE: (removed intermediate lines)

```
[switch0:initialize] Initializing switch.
[switch0:initialize] Connected nodes: 1 2 3
[switch0:generateMessage] Generated new message 'msg' from 0 to 11.
[switch1:initialize] Initializing switch.
[switch1:initialize] Connected nodes: 0 2
[switch1:initializationMessage] Initializing switch message.
[slave3:initialize] Initializing slave controller.
[domain0:initialize] Initializing domain controller.
[domain0:getIsIn] Number of switches under control: 0.
[domain0:retrieveCondition] Retrieving condition from message ''.
[domain0:getIsIn] Number of switches under control: 7.
[domain0:forwardMessageToSlave] Forwarding message 'request' to slave controllers. Src = 0, Des = 11.
[domain1:handleMessage] Received message 'request' from 'super’
[domain1:handleMessage] Forwarding retrieve request with condition to super controller.
[super:retrieveCondition] Retrieving condition from message 'retrieve'.
[super:handleMessage] Received 'retrieve' message. get = 1.
[super:retrieveCondition] Retrieving condition from message 'retrieve'.
[super:handleMessage] Received 'retrieve' message. get = 2.
[super:handleMessage] All condition messages received. Running SARSA.
[super:getTau] visit[0][11] = 1, computed tau = 99.100000.
propability of 1 value: 0.333333
propability of 2 value: 0.333333
propability of 3 value: 0.333333
[super:getReward] For state 4 to nextState 9, reward = -1.534165.
[super:sarsa] state 4, action 9, nextState 9, nextAction 11, reward -1.534165.
[super:sarsa] Updated Q[4][9] = -1.534165.
[super:getTau] visit[0][11] = 1, computed tau = 99.100000.
propability of 9 value: 0.5
propability of 12 value: 0.5
[super:getReward] For state 9 to nextState 11, reward = 2.963072.
[super:sarsa] state 9, action 11, nextState 11, nextAction 12, reward 2.963072.
[super:sarsa] Updated Q[9][11] = -2.963072.
[super:sarsa] Completed SARSA for route from 0 to 11. visit[0][11] = 1.
[super:handleMessage] Calculated route: 0 -> 3 -> 4 -> 9 -> 11
```

SARQ – cube of various controllers







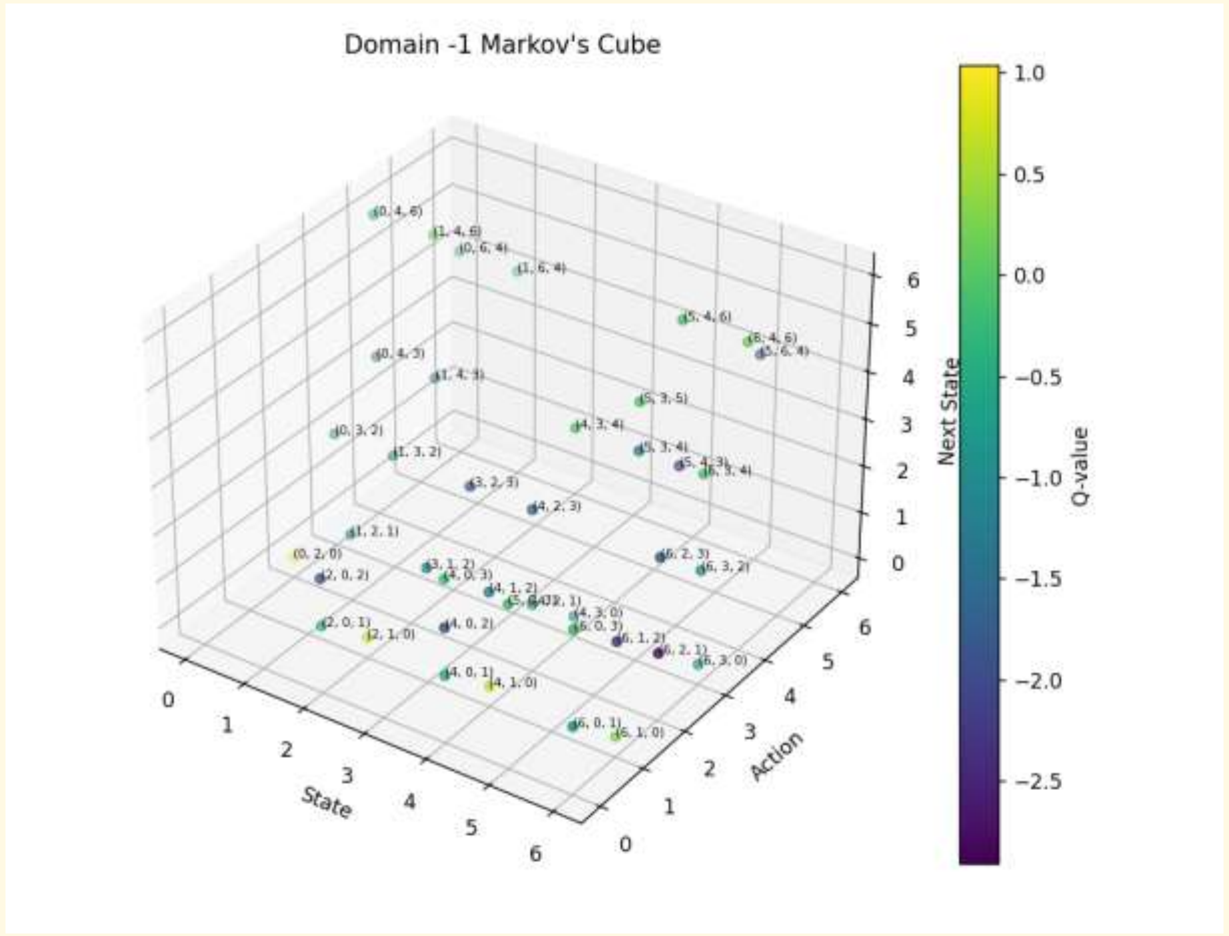
Super controller

Domain 1 controller

Domain 2 controller

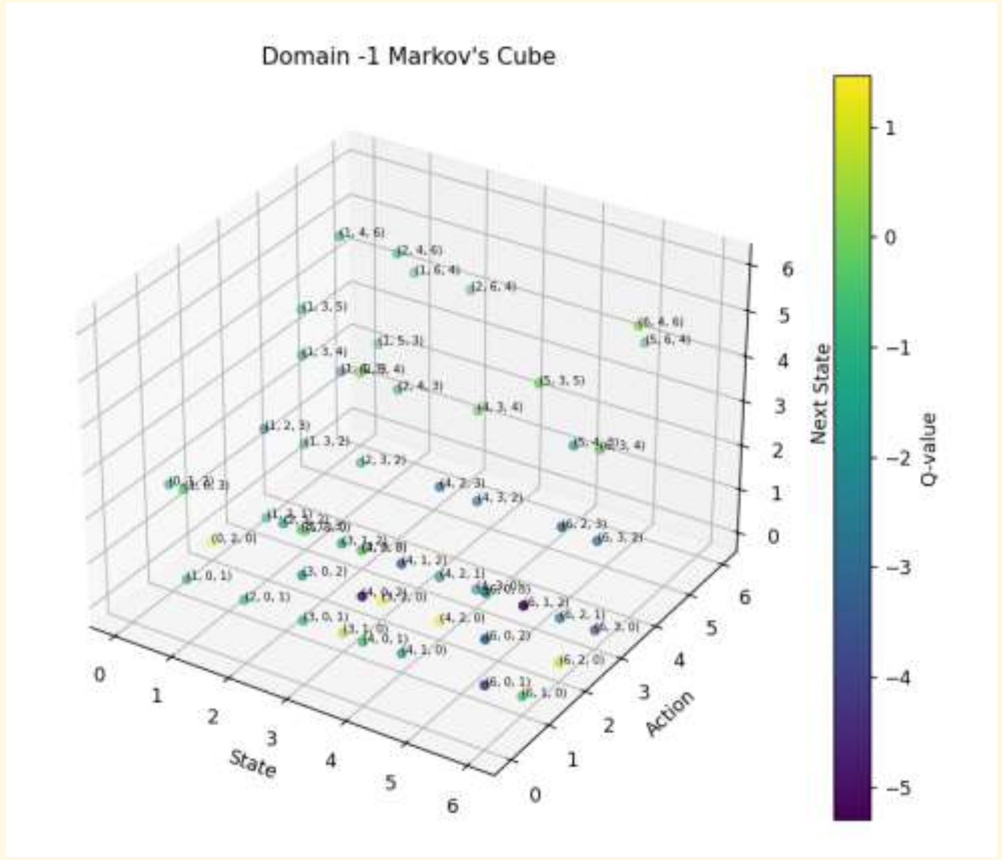
Alpha (learning rate) = 0.9
Discount factor (gamma) = 0.7

Variable weights and their impacts on SARQ - cube



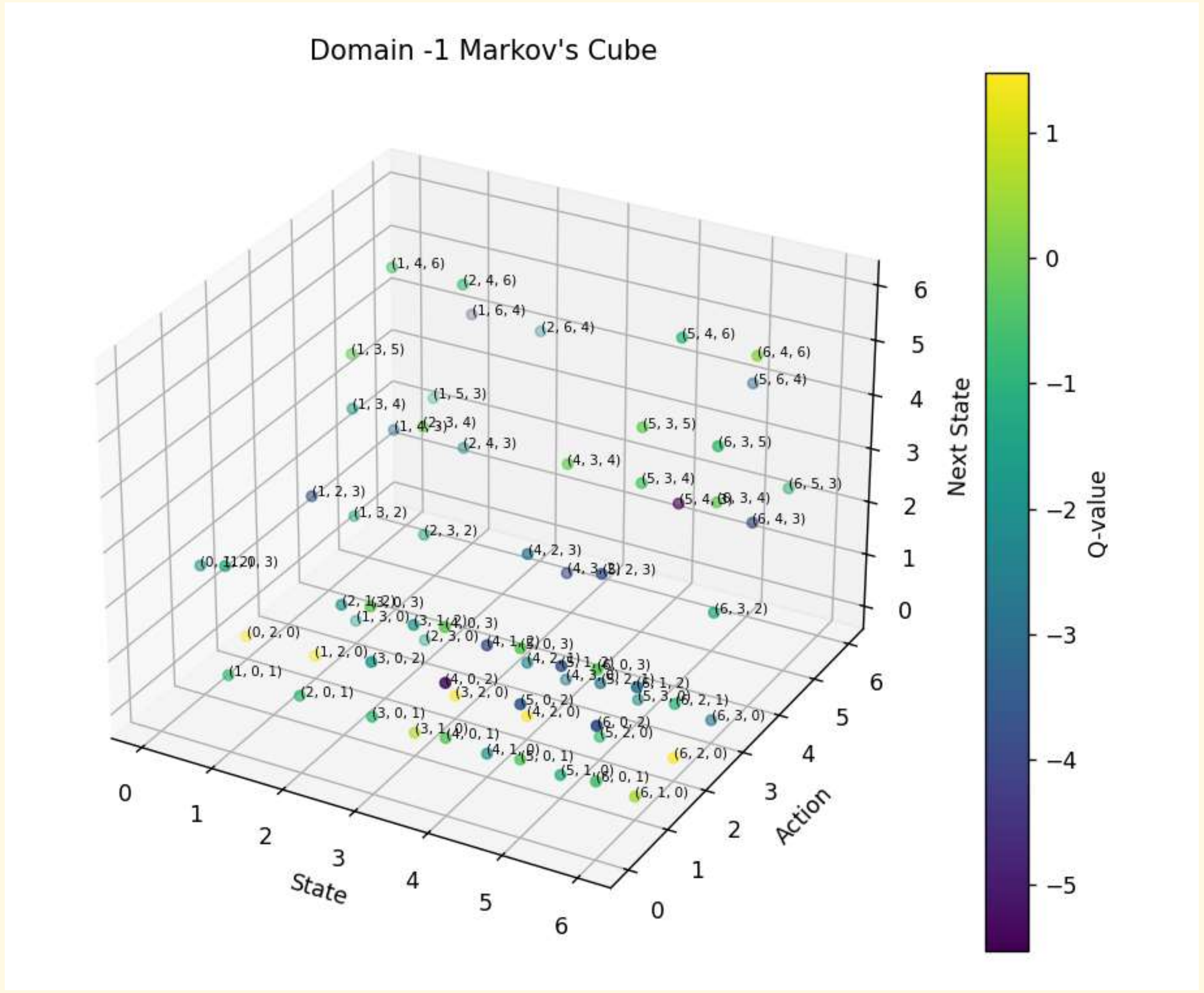
Variable Alpha = 0.7
Fixed gamma = 0.7

Variable weights and their impacts on SARQ - cube



Variable Alpha = 0.9
Fixed gamma = 0.7

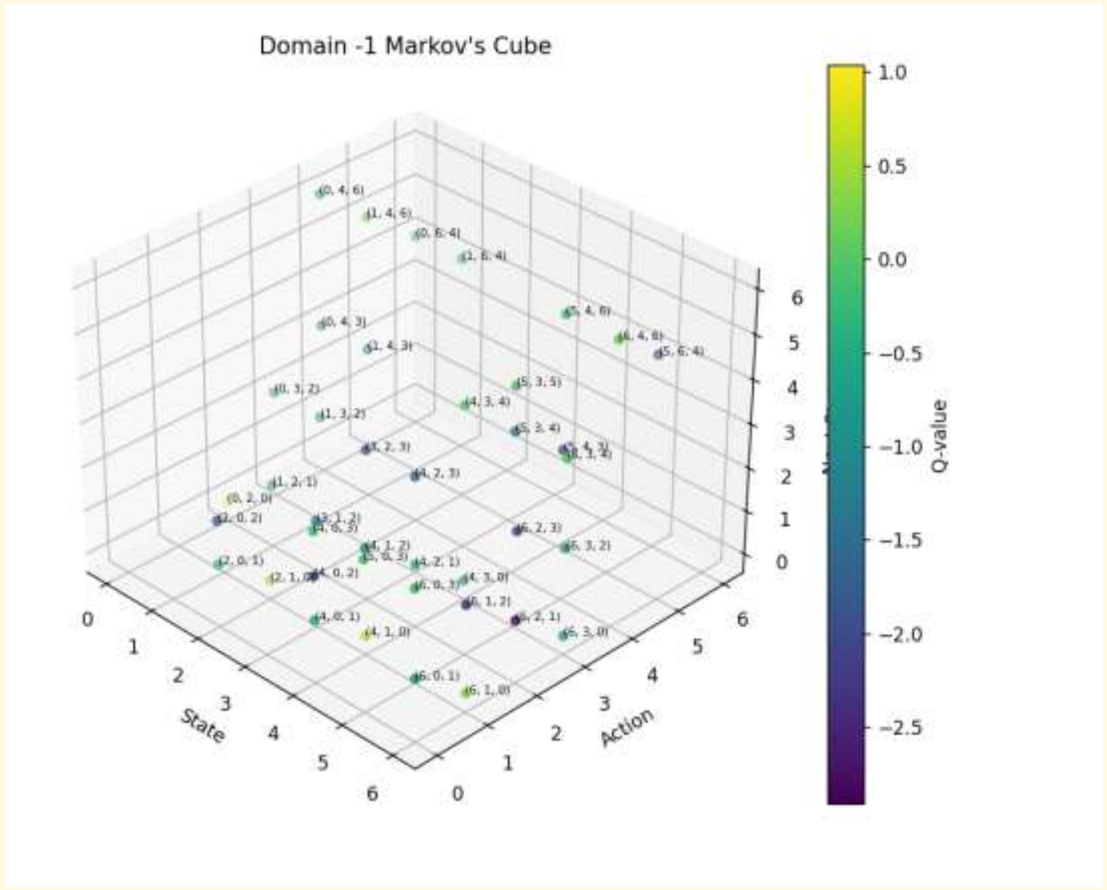
Variable weights and their impacts on SARQ - cube



Variable Alpha = 1
Fixed gamma = 0.7

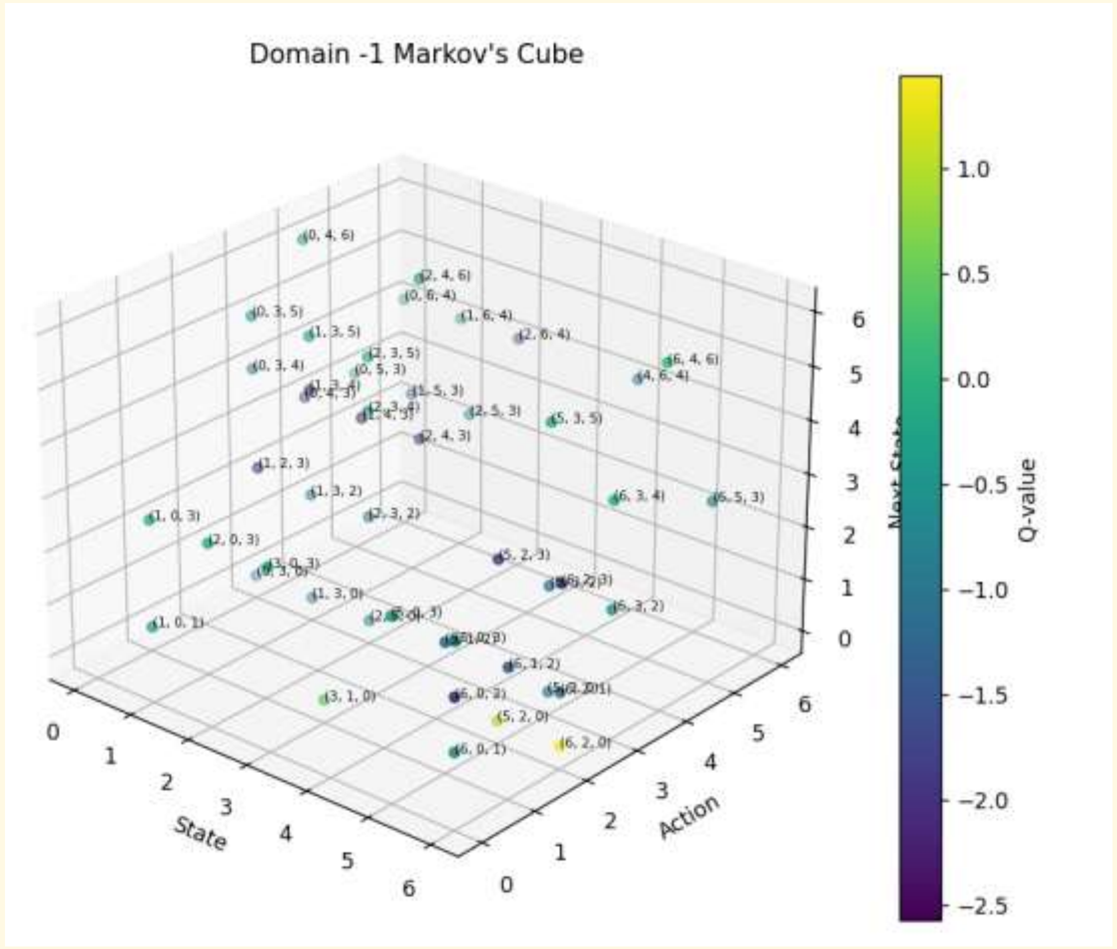
(attains learning fatigue -> attains learning convergence)

Variable weights and their impacts on SARQ - cube



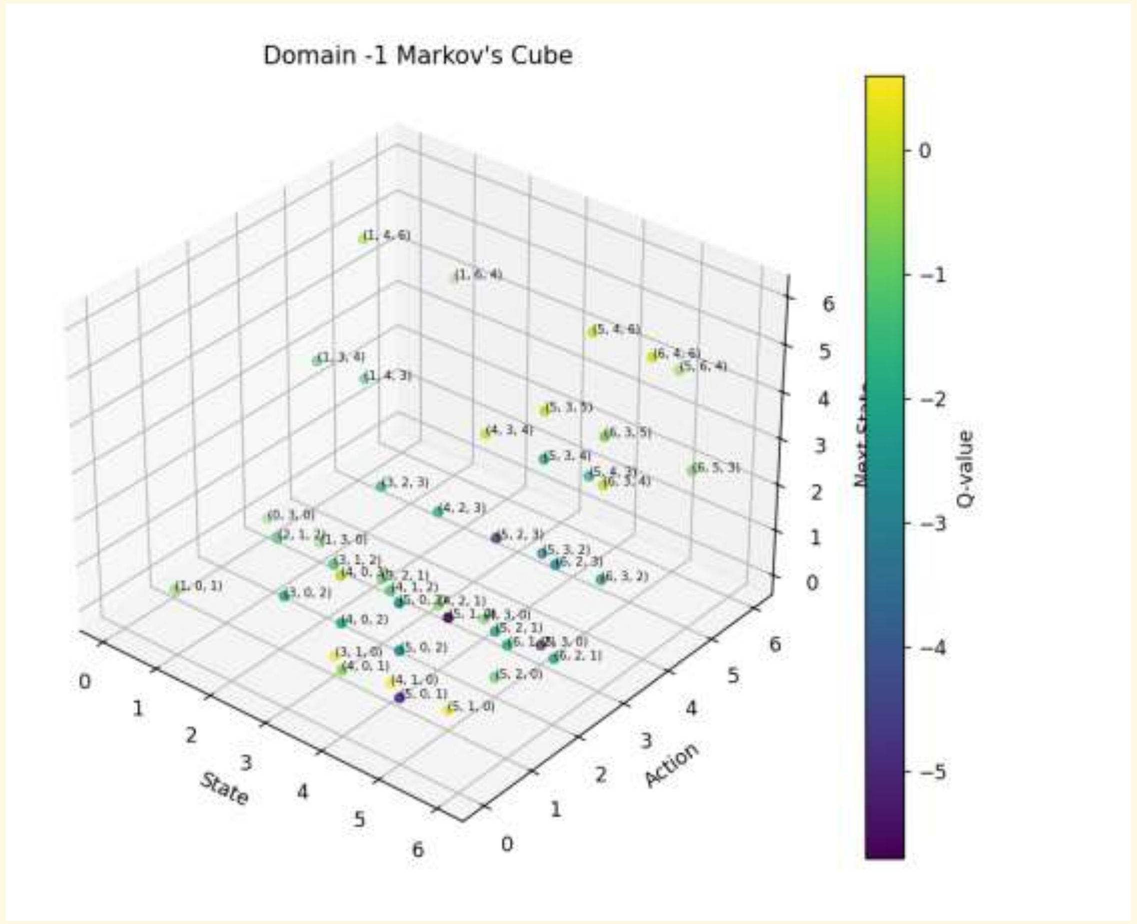
Fixed Alpha = 0.7
Variable gamma = 0.7

Variable weights and their impacts on SARQ - cube



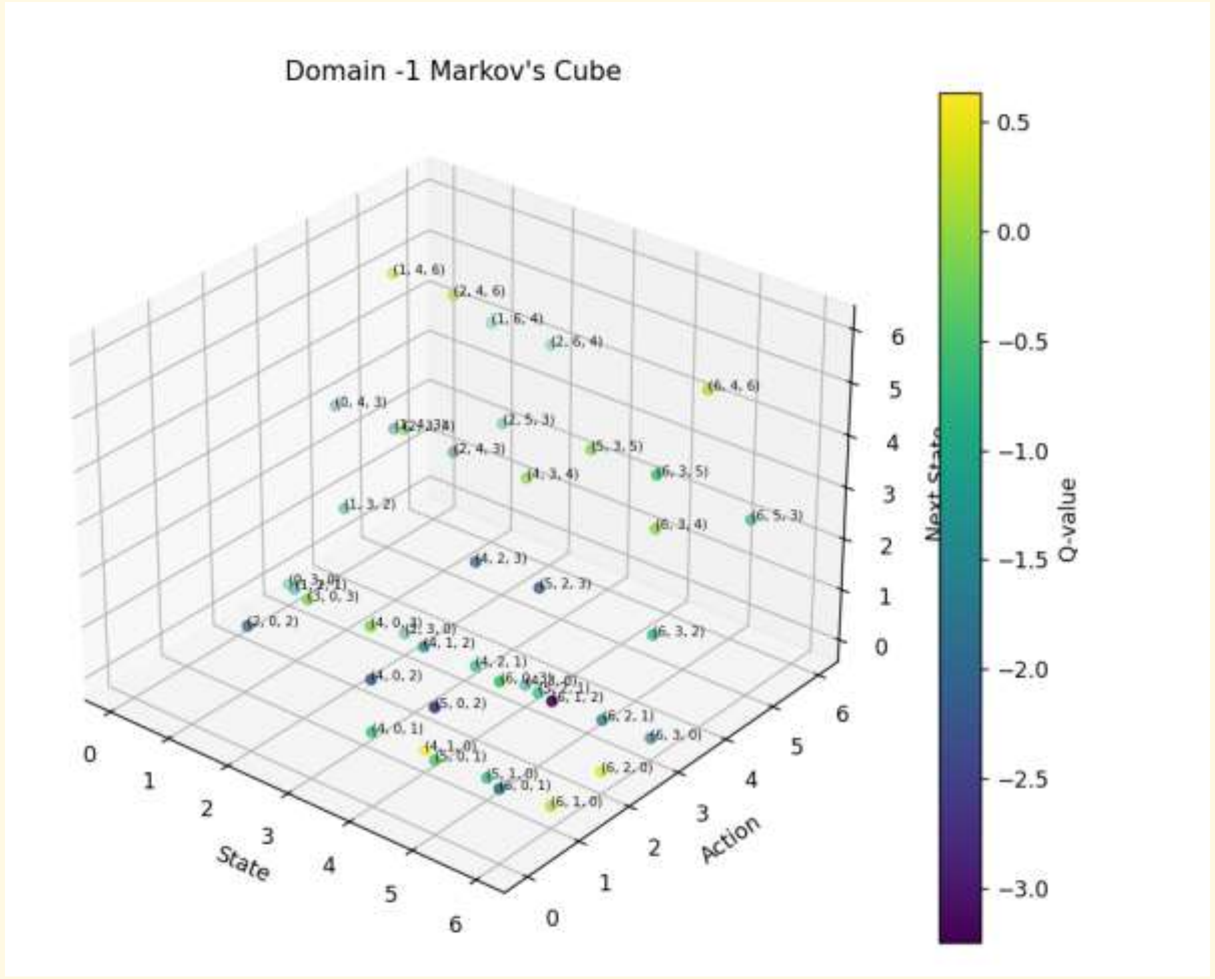
Fixed Alpha = 0.7
Variable gamma = 0.8

Variable weights and their impacts on SARQ - cube



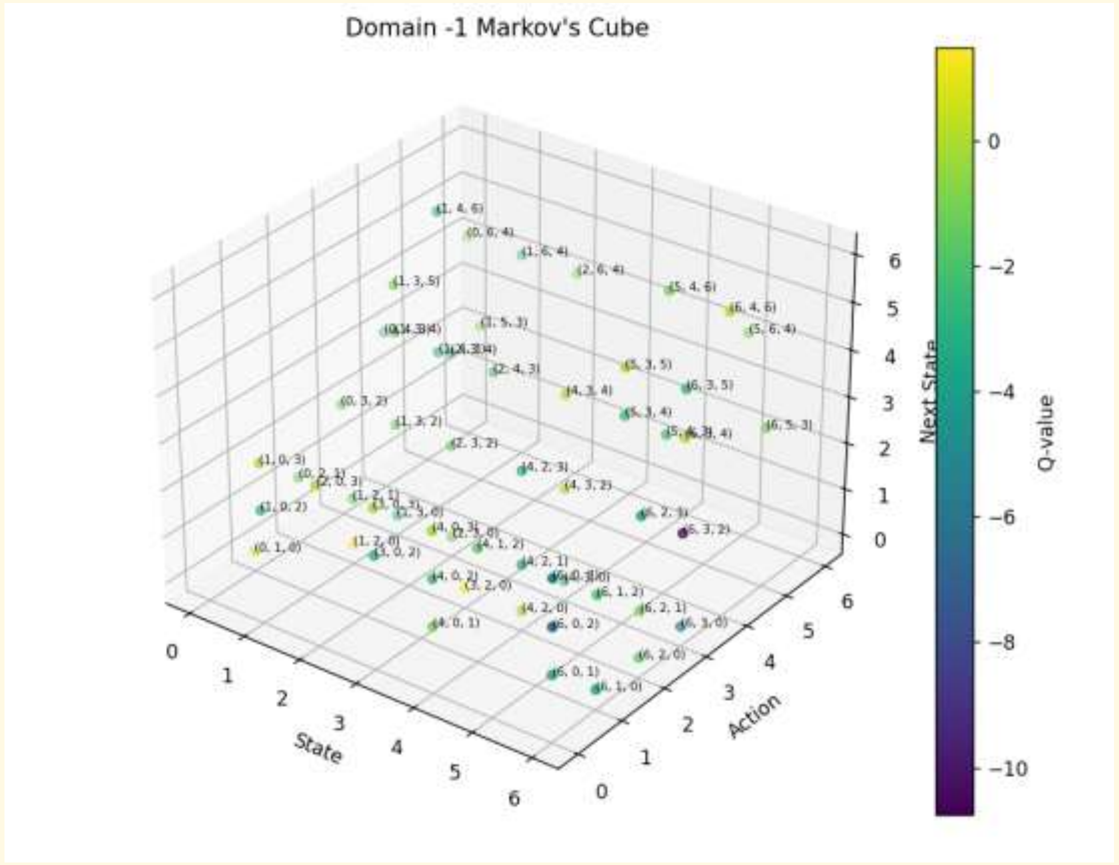
Fixed Alpha = 0.7
Variable gamma = 1

Variable weights and their impacts on SARQ - cube



Variable Alpha = 0.8
Variable gamma = 0.8

Variable weights and their impacts on SARQ - cube

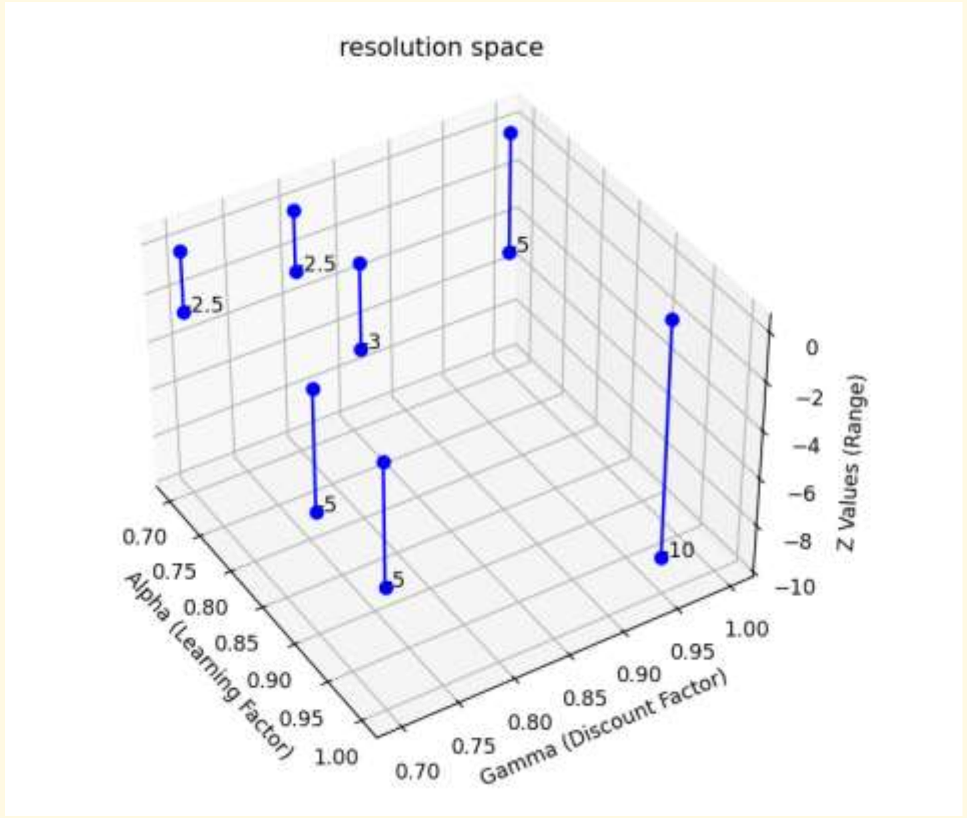


Variable Alpha = 0.97
Variable gamma = 0.97

(SARSA learns towards → Epsilon Greedy)

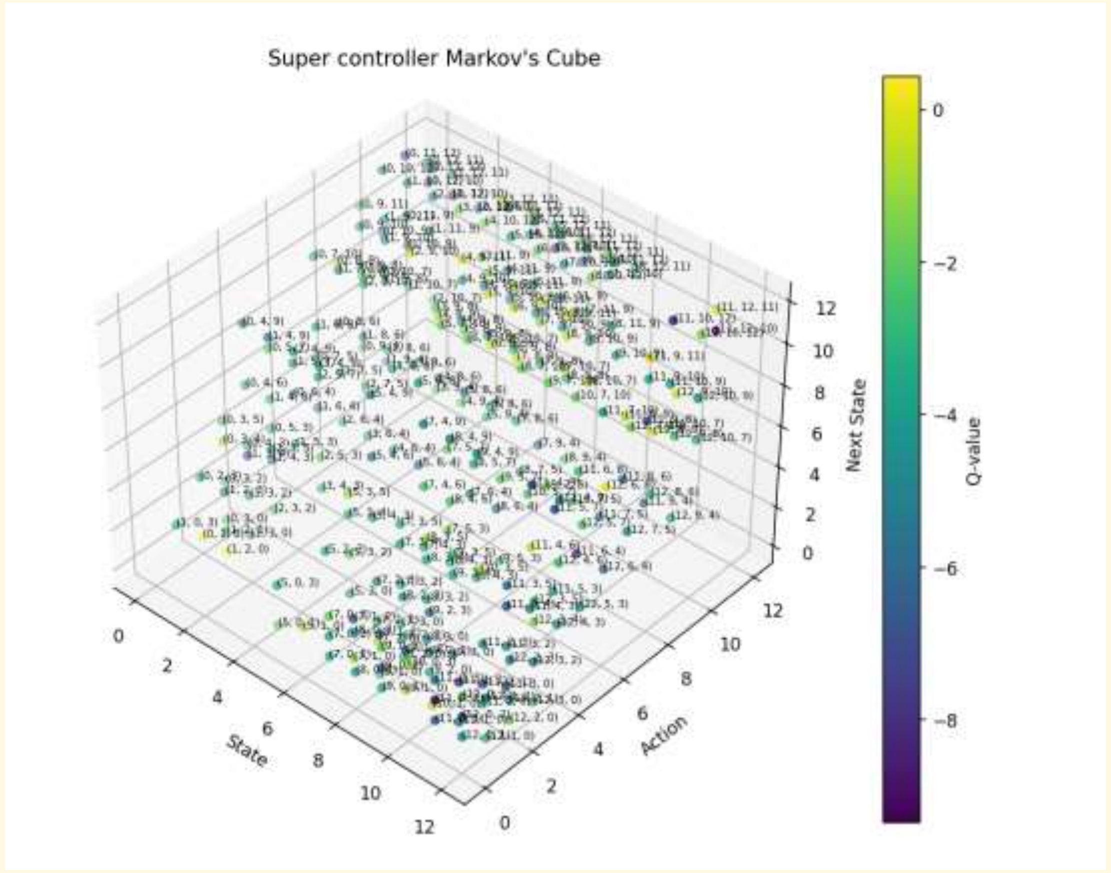
Resolution space

Plot of α , γ and resolution values
To demonstrate learning saturation

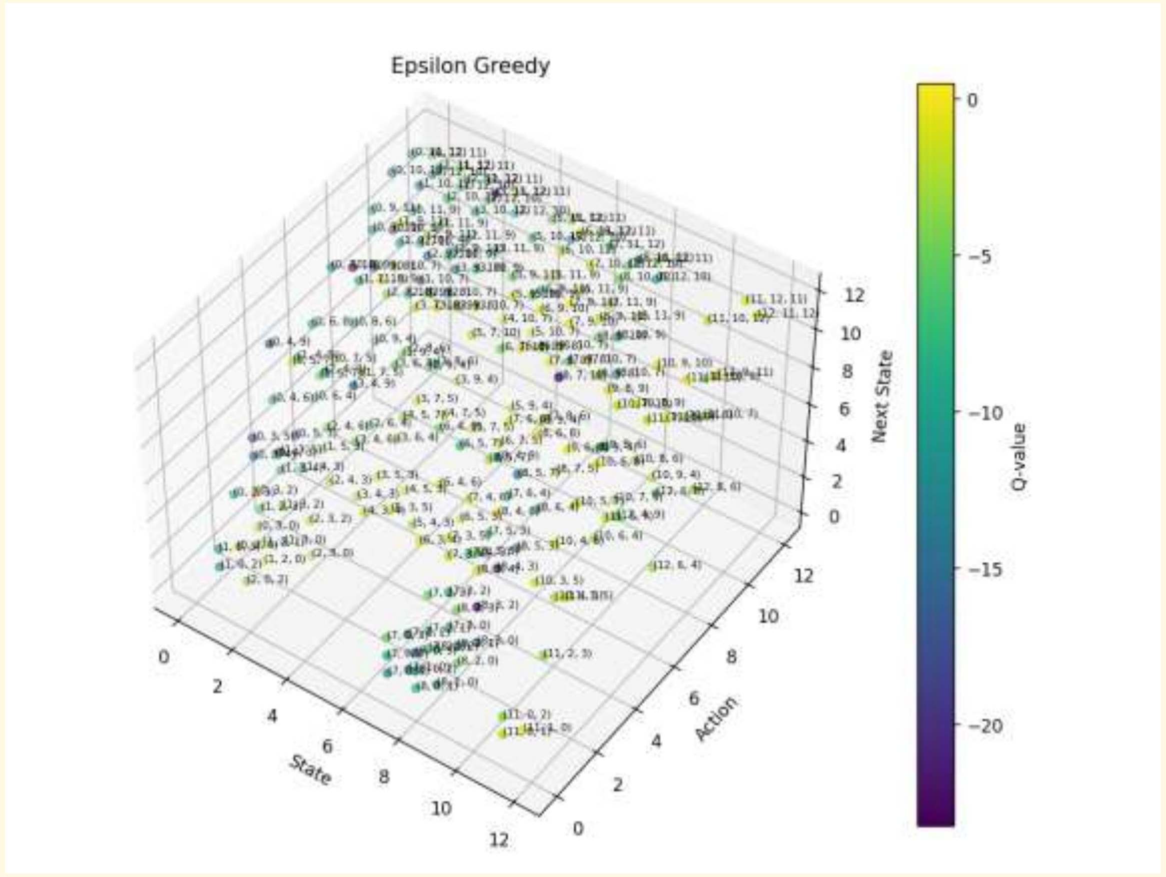


FINAL RESULTS

SARSA with **α** = 0.7 **γ = 0.7**



Off policy Q –learning with **α** = 0.99 **γ = 0.99**



Hybrid Reinforcement Learning (Hybrid RL)

Hybrid Reinforcement Learning (Hybrid RL) combines multiple RL techniques, algorithms, or paradigms to leverage their strengths while mitigating individual weaknesses. It typically integrates:

1. Model-Free & Model-Based RL – Using model-free methods like SARSA or Q-learning for policy learning while incorporating model-based planning for faster convergence.

2. On-Policy & Off-Policy Learning – Mixing algorithms like SARSA (on-policy) with Q-learning (off-policy) to balance exploration and exploitation.

3. Supervised/Unsupervised Learning with RL – Incorporating neural networks (deep learning) or clustering methods to enhance decision-making.

4. Multi-Agent RL (MARL) & Single-Agent RL – Combining cooperative and competitive learning in multi-agent systems.

5. Meta-Learning in RL – Adapting policies dynamically based on past experiences

Transferring of weights :

Instead of initializing the Q-table randomly or setting all values to zero, the system reuses past training data:

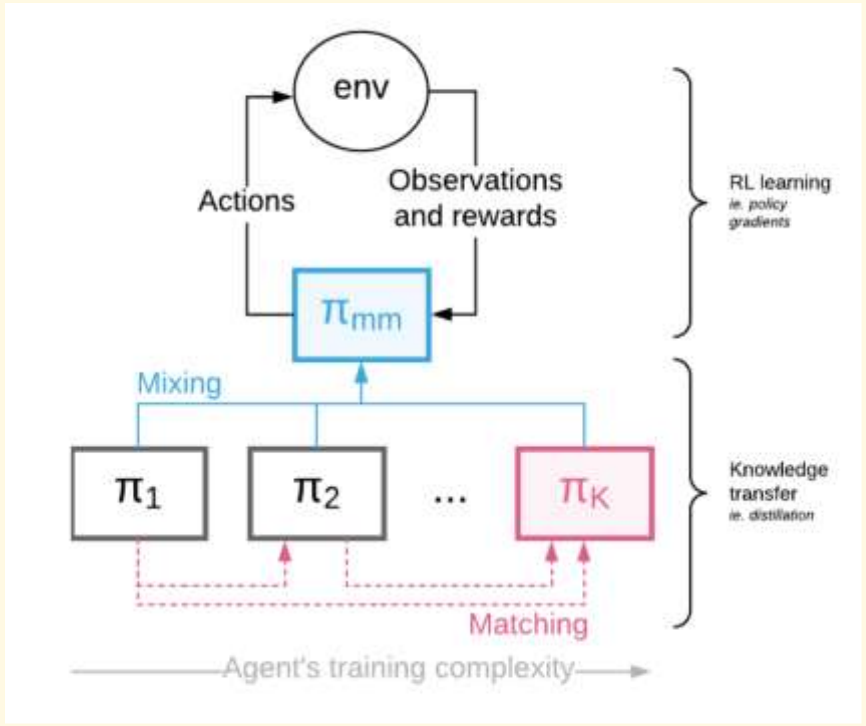
$Q(s,a) \leftarrow Q_{prev}(s,a)$

where:

$Q_{prev}(s,a)$

is the Q-table from the previous training iteration.

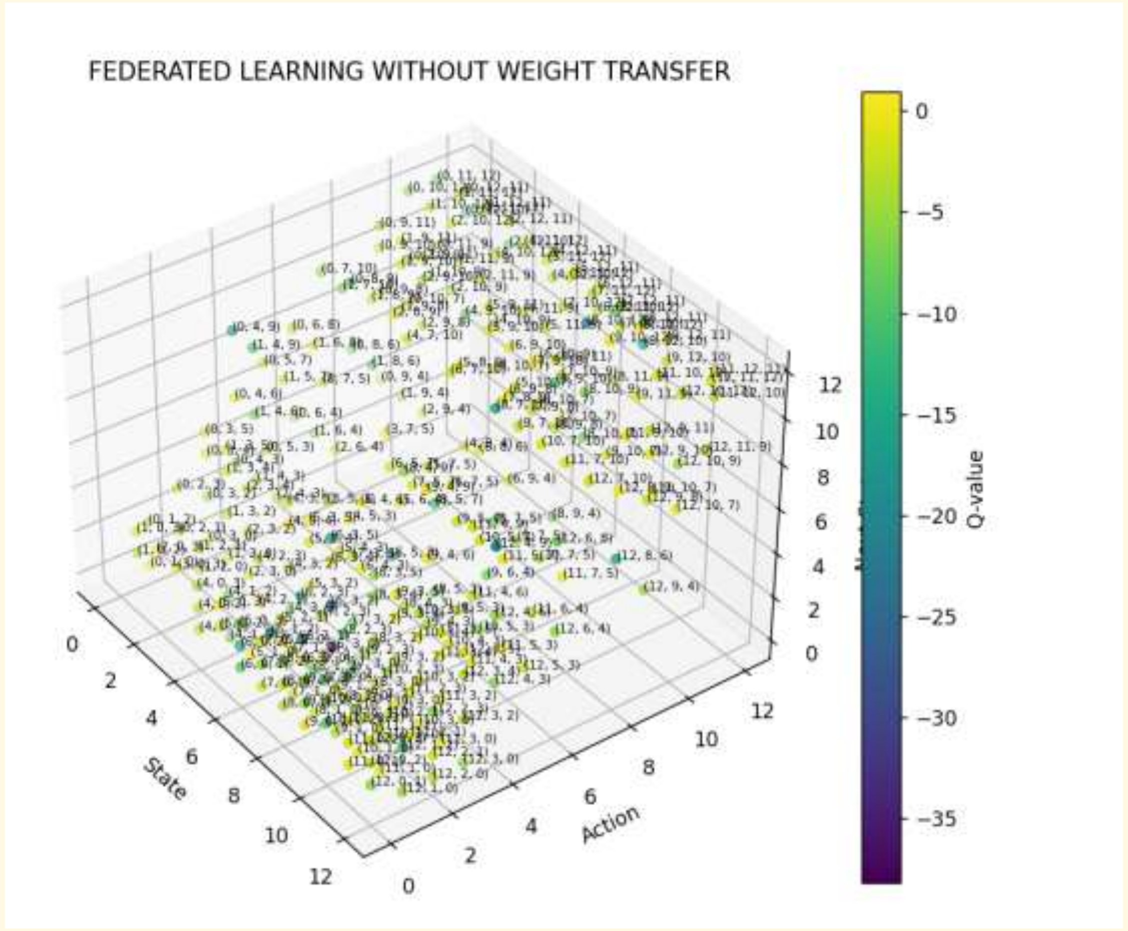
This allows knowledge transfer from previous runs, avoiding the need to relearn already optimized paths.

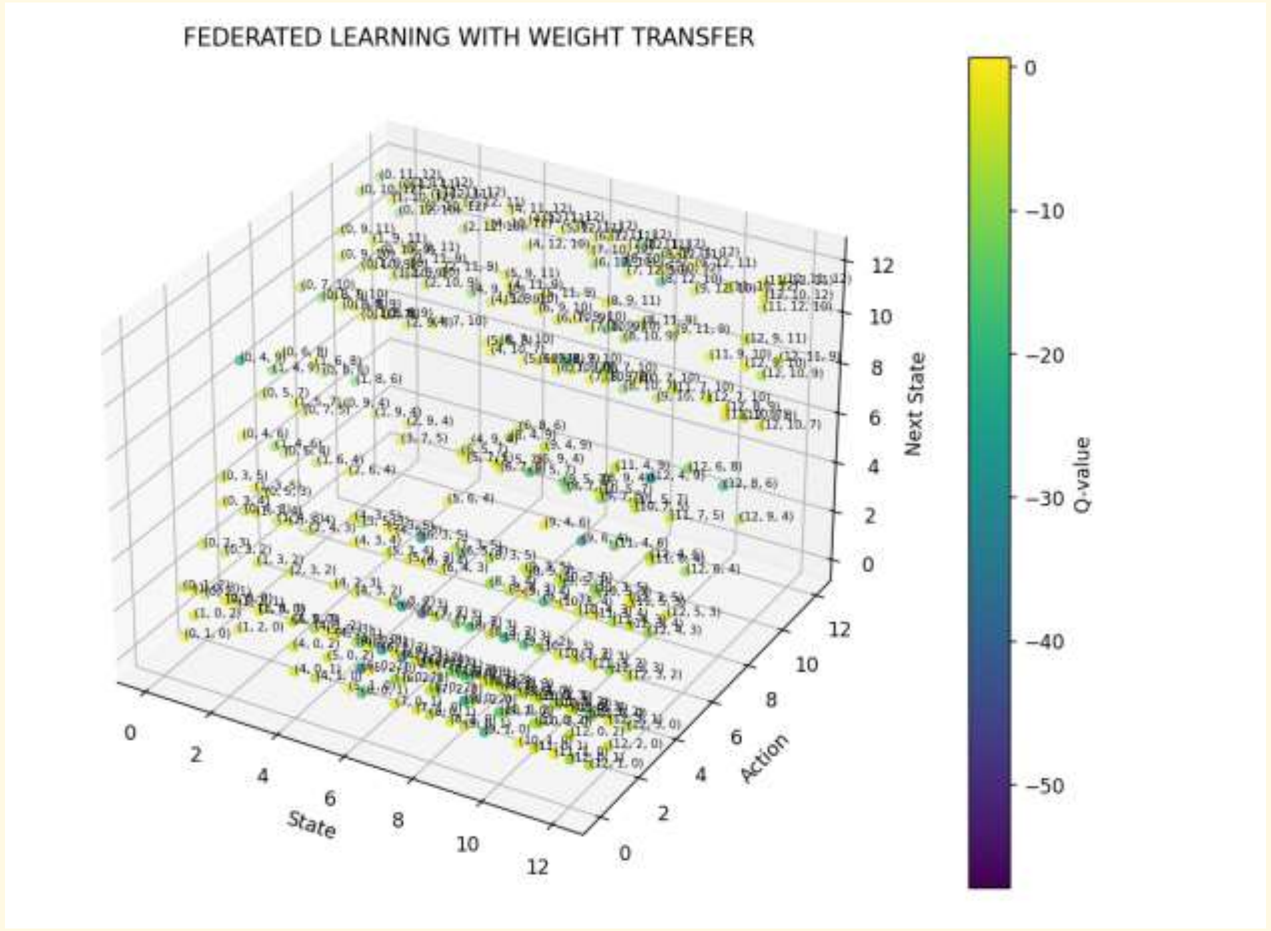


FINAL RESULTS

Multi agent Hybrid Federated Learning results of super controller

Hybrid Federated Learning with pre-trained weight initialization of previous iterations to mimic transferred Federated learning





THANKYOU